Automated, FPGA-Based Hardware Emulation of Dynamic Frequency Scaling

by

David Thomas Biancolin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Co-chair
Adjunct Assistant Professor Jonathan Richard Bachrach, Co-chair
Professor Sanjit Seshia
Professor Robert Leachman

Spring 2021

Automated, FPGA-Based Hardware Emulation of Dynamic Frequency Scaling

Abstract

Automated, FPGA-Based Hardware Emulation of Dynamic Frequency Scaling

by

David Thomas Biancolin

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Krste Asanović, Co-chair

Adjunct Assistant Professor Jonathan Richard Bachrach, Co-chair

The simultaneous growth of new applications and death of transistor scaling trends is driving an explosion in custom silicon projects spanning all domains of computing. However, the enormous non-recurring engineering (NRE) cost of designing a modern system-on-a-chip (SoC) remains a major barrier to the wider adoption of custom silicon. Of concern to this dissertation is the lack of a good full-system simulation technology, a key driver of pre-silicon verification and validation costs. While field-programmable gate arrays (FPGAs) can be fast and relatively inexpensive hosts for simulation, mapping SoC clocking structures onto an FPGA such that they are represented accurately and deterministically is challenging. For this reason and others, many SoC designers turn to expensive hardware emulation platforms and their proprietary compilers. To radically reduce the cost of doing fast and accurate full-system simulation, the ADEPT Lab designed FireSim: an open-source, FPGA-based hardware emulation framework hosted in the public cloud.

In this dissertation, we begin by introducing FireSim's compiler infrastructure, called Golden Gate, which is capable of performing general multi-cycle resource optimizations in order to fit large SoCs on a single FPGA. Here we extend Golden Gate to present non-invasive, optimization-compatible schemes for simulating SoC clocking structures. First, we describe a simple approach for simulating systems with multiple fixed-frequency clocks. We then generalize this to support a general class of clock and reset structures, which can be composed to simulate dynamic frequency scaling, using an approach based on prior work in conservative parallel discrete-event simulation. The resulting work differs from prior academic FPGA-based projects in that it supports a much larger space of input designs, is easier to deploy to other FPGAs, and, like any good simulator, is deterministic—all while supporting simulation rates fast enough to productively boot operating systems and run real applications.

For Ivan Blunno (1973 - 2016)

Contents

Contents **ii**

List of Figures **vi**

List of Tables **vii**

Glossary of Terms **viii**

Acknowledgements **ix**

1 Introduction **1**
1.1 Previous Publication, Collaboration, and Funding 6

2 Full-System Simulation of SoCs **7**
2.1 A Tour of Full-System Simulation for SoC Design 7
2.1.1 CPU-Hosted Simulation for Prototyping 8
2.1.2 CPU-Hosted Simulation for Verification 8
2.1.3 FPGA Prototyping 9
2.1.4 Hardware Emulation 10
2.2 Hardware-Accelerated Microarchitecture Simulation 12
2.2.1 Research Accelerator For Multiple Processors (RAMP) 12
2.2.2 FAME Taxonomy 13
2.2.3 FAME-1: Host Decoupling 13
2.2.4 FAME-2: Abstract RTL 14
2.2.5 FAME-4: Multithreading 14
2.2.6 RAMP Retrospective 15
2.3 Recent Work at Berkeley 15
2.3.1 Strober and MIDAS 16
2.3.2 FireSim 18
2.4 Motivations for Golden Gate (MIDAS II) 19

3 On The Design of FPGA-Based Discrete-Event Simulators **20**
3.1 An Iron Law For FPGA-Based Simulator Performance 21

3.2 Centrally Controlled (CC) Simulators 22
3.3 Distributed Control (DC) Simulators 24
3.3.1 A Primer on Parallel Discrete-Event Simulation 24
3.3.2 Considerations for FPGA-Hosted PDES 25
3.3.3 ASIC Emulation As Domain-Specific PDES 26
3.3.4 Target Formalisms for ITDC Simulators 27
3.3.5 Channel-Bootstrapped Formalisms 27
3.3.6 Latency-Insensitive Bounded-Dataflow Networks (LI-BDN) 29
3.4 The MIDAS Compiler & Generated Simulator Microarchitecture 35
3.4.1 Endpoint Units 36
3.4.2 Simulation Driver 38
3.4.3 Time Control 38
3.4.4 MIDAS's Generator Side Integration 39
3.4.5 The MIDAS Compiler Flow 39
3.5 Reviewing MIDAS's Limitations 42

4 Golden Gate: An Optimizing FAME Compiler 43
4.1 Design Objectives 43
4.2 Compiler Input & Target Specification 45
4.3 Updated Compiler Flow 46
4.3.1 Annotations 49
4.4 Target Transformation 50
4.5 Simulator Synthesis 51
4.6 Platform Mapping 52
4.6.1 Channel Synthesis 52
4.6.2 Bridge Instantiation 53
4.6.3 Platform Wrapper 54
4.7 ICCAD 2019 Golden Gate Publication 54

5 Support for Systems With Multiple Fixed-Frequency Clocks 56
5.1 Implementation 58
5.1.1 Simplifying Assumptions 58
5.1.2 Annotation Modifications 59
5.1.3 Modified FAME Transform 59
5.1.4 Clock Bridge 61
5.1.5 Other Compiler & Bridge Modifications 62
5.1.6 The Base Clock 62
5.1.7 Rethinking Simulation Performance 63
5.2 Case Study - SPEC CPU 2017 Integer Performance 64
5.3 Scaling Target Clock Count 69
5.4 Improving Simulator Performance with Multi-Cycle Setup Constraints 71
5.5 Future Work 72

6 Distributed Simulation of Clocking Structures via Explicit Timekeeping 74
6.1 Context, Goals and Motivation . . . 75
6.2 Other Designs We Considered . . . 76
6.3 Implementation Overview . . . 77
6.4 Simplifying Assumptions . . . 79
6.5 Target-Side User Interface . . . 79
6.6 Message Representation and Channel Design . . . 80
6.7 Correctness Of Logical Processes . . . 81
6.8 Hub Unit Modifications . . . 82
6.9 Compiler and Annotation Modifications . . . 84
6.10 Baseline Timestamped Units . . . 86
6.10.1 Timestamped Tuples: Unpacked Messages . . . 87
6.10.2 Edge-Triggered D Flip-Flops . . . 88
6.10.3 Generic Combinational Functions . . . 89
6.10.4 Fan Outs . . . 89
6.10.5 Clock and Asynchronous-Reset Sources . . . 90
6.10.6 Library Timestamped Units . . . 90
6.11 Verification . . . 91
6.12 Performance in Common Clock Organizations . . . 92
6.12.1 Feedforward Divider Networks . . . 93
6.12.2 Simple Hub Cycles . . . 93
6.12.3 Multi-Unit Cycles . . . 94
6.12.4 Compositions Of Circuits . . . 95
6.13 Lookahead Optimized Units . . . 95
6.13.1 Integer Clock Dividers . . . 95
6.13.2 Integrated Clock-Gating Cells . . . 96
6.13.3 Clock Multiplexers . . . 97
6.14 FPGA QoR And Scalability . . . 98
6.14.1 TU Resource Utilization . . . 98
6.14.2 Hub Unit QoR and f_{max} Scaling . . . 99

7 Full-System Case Studies of SoCs Deploying Frequency Scaling 101
7.1 Thermal Regulation via Dynamic Frequency Scaling . . . 101
7.1.1 Target SoC Description . . . 102
7.1.2 System Software Description . . . 103
7.1.3 Demonstration . . . 104
7.2 Latency Overlapping in Resource-Optimized BOOM-Based Simulators . . . 105

8 Limitations and Future Work 108
8.1 Performance: Overcoming Throughput Bounds . . . 108
8.2 Performance: Overcoming Latency Bounds . . . 109
8.3 Future Work, Performance Aside . . . 110

9 Conclusion **112**

Bibliography **114**

List of Figures

1.1 Early design NRE at various feature dimensions. Source: IBS. 3

3.1 Two types of channels used in a channel-bootstrapped formalism. 28
3.2 A 32-bit adder model and environment simulating two cycles of target time. . . 30
3.3 Three channel-modeling decisions for latency-insensitive target interconnect. . . 31
3.4 A wrapper-module-based conversion of a patient SSM into a primitive LI-BDN. 34
3.5 A wrapper-module-based conversion of a patient SSM into an APort Network unit. 35
3.6 The graph of a typical Rocket-Chip-derived system. 36
3.7 An example simulator mapped to an Amazon EC2 F1 host (`2x.large`). 37
3.8 The end-to-end flow of a MIDAS compilation. 40

4.1 A typical input Chipyard SoC and its associated Golden Gate annotations. . . . 47
4.2 The operation of core phases of the Golden Gate compiler. 48
4.3 The wrapper circuit generated by platform mapping with channels and bridges. 53
4.4 The LIME model checking flow. 55

5.1 A wrapper-module-based conversion of multi-clock target RTL into a unit. . . . 60
5.2 An example encoding of three rationally related clocks. 61
5.3 Per-iteration execution latency of a randomized pointer chase versus array size. . 66
5.4 SPEC2017 intspeed performance on Rocket versus different clock organizations. 67
5.5 SPEC2017 intspeed performance on BOOM versus different clock organizations. 68

6.1 A simulator graph with timestamped and untimestamped subgraphs. 78
6.2 The wrapper module for the hybrid-timestamped hub unit. 83
6.3 A visualization of the promote passthrough optimization. 85
6.4 Reference circuit diagrams for the baseline TUs. 91
6.5 Five illustrative clock organizations that leverage each of the TUs we have built. 92
6.6 An illustration of the clock gate configuration's measured FMR of 6.0. 94
6.7 A synchronized, glitchless 2:1 multiplexer that can switch unrelated clocks. . . . 97
6.8 Measured f_{fpga} versus timestamped input count of a 1-bit-wide scan-chain. . . . 99
6.9 LUT utilization of the simulation wrapper versus timestamped input count. . . 100

7.1 The modified, single-tile target used to illustrate thermal throttling. 102
7.2 IPC and temperature versus simulated time of the thermal-throttling case study. 104

List of Tables

5.1 Microarchitectural parameters of the two targets under test. 65
5.2 Dynamic instruction counts and L1 MPKIs of SPEC2017 intspeed benchmarks. 67
5.3 SPEC 2017 Integer Speed (single-threaded) emulation performance. 69
5.4 Simulator f_{max} change using multi-cycle setup constraints. 72
5.5 Simulator performance change using multi-cycle setup constraints. 72

6.1 Measured FMR of the clock organizations of Figure 6.5 using baseline TUs. . . . 93
6.2 Measured FMR of the clock organizations of Figure 6.5 using optimized TUs. . . 96
6.3 TU resource utilization when targeting a Xilinx Ultrascale+ device. 98

7.1 Measured FMR of a BOOM-based SoC using resource optimizations and TUs. . 106

Glossary of Terms

Abbreviations

ASIC	Application-Specific Integrated Circuit
FAME	FPGA Architecture Model Execution
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
LI-BDN	Latency-Insensitive Bounded Dataflow Network
LP	Logical Process
NRE	Non-Recurring Engineering [cost]
PDES	Parallel Discrete-Event Simulation
SoC	System-on-a-Chip. Used interchangeably with ASIC in this text.
RTL	Register Transfer Level

Key Definitions

Bridge	A target black-box and its custom unit implementation.
Chisel	An open-source HDL embedded in Scala
Channel	A point-to-point LP that may model stateful interconnect.
FIRRTL	Flexible Intermediate Representation for RTL
Host Decoupling	The property that permits simulated time to advance independently of wall-clock (host) time. Simulators are host-decoupled. FPGA prototypes are not.
Host	The system on which a simulation executes.
Host Time	Wall-clock time. Seen by the user and the host.
Hub Unit	The singleton unit representing all unoptimized RTL.
Message	A 2-tuple consisting of a signal value and a timestamp.
Model	An optimized UU. A valid primitive LI-BDN.
Target	The closed system under simulation.
Target Time	The time perceived by the system under simulation. See Target.
Token	An untimestamped data value.
TU	A timestamped unit; a unit that acts on messages.
Unit	An LP representing a part of an SoC or its environment.
UU	An untimestamped unit; a unit that acts on tokens.

Acknowledgments

In normal times, a Ph.D. is a long and often solitary journey, one made possible by many people not represented in the author lists of talks or publications. COVID-19 has put this fact on full display. Suffice it to say, the past year has given me time in abundance to reflect on those who have made my Ph.D. possible, its difficult times endurable, and its good times wonderful.

I must start by thanking the first of my two advisors, Jonathan Bachrach, whose dynamism had a huge role to play in attracting me to Berkeley in 2014. It's been a pleasure to work with him on a variety of projects over the years, including JITPCB. Jonathan has an unbounded imagination for research and his excitement to realize it is truly unmatched. Over the years I've tried to import at least some of that into my own research. In my final year of graduate school, Jonathan was instrumental in seeing me through to the end.

During visit days, I remember telling one U of T faculty member I was interested in working with my future second advisor, Krste Asanović, to which he replied: "Oh, he likes to build stuff." This remains a fair characterization—Krste and his students did roll their own ISA after all. It was Krste's, and by osmosis his group's, desire to build and study real systems that made it so exciting to be at Berkeley the past seven years. His technical advice has always been invaluable and, despite being ludicrously oversubscribed in recent years, Krste has been there when I needed him.

I must thank Sanjit Seshia, whose course on formal verification (EE219C) was the most impactful class I took in graduate school. It changed how I thought about verification, and made me appreciate how formal tools could be wielded as a practical tool for finding bugs. It also gave me the opportunity to convince my friend and considerably more talented colleague, Albert Magyar, to work on FireSim stuff with me. Indeed, EE219C's class project was the genesis of Golden Gate, and Sanjit's and Pramod Subramanyan's guidance in those early days were crucial.

FireSim is a large project made possible by contributions from many people, including Abe Gonzalez, Albert Ou, Sarah Zhou, Jerry Zhao, Nathan Pemberton, Dayeol Lee, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Tim Snyder, and others still. I'd like to particularly acknowledge the roles of Howard Mao, whose ability to jump into a gnarly codebase and get things working has always been source of inspiration for me; Donggyu Kim, with whom I initially collaborated on MIDAS; Alon Amid, for his tenacity in attempting to make our tools usable outside of Berkeley and for championing the Chipyard project; and Sagar Karandikar, who has carried FireSim on his back, and whose engineering and leadership sensibilities have strongly informed my own. It's been an enormous pleasure to build out FireSim with you all.

FireSim is one organism in a larger ecosystem of research underway in the UC Berkeley Architecture Research Group (UCB-BAR). I'm deeply indebted to the students who've passed through the group during my time at Cal. Many of the students that preceded me, including Adam Izraelevitz, Ben Keller, Scott Beamer, Henry Cook, Yunsup Lee, Chris Celio, Palmer Dabbelt, Brian Zimmer, John Wright, Angie Wang, and Paul Rigge, have mentored me either explicitly or unwittingly, and for that I am grateful. For many years, Eric Love in particular went above the call of duty, and took on the role of organizer-in-chief of Krste's group, for this he deserves a ton of credit. Lisa Wu Wills, one of the rare post-docs in the group, was always willing to mentor students. I'm thankful for the time she took to coach me through my first paper presentation. David Bruns-Smith, a fellow climber, skier, and MTGer, was reliably down to walk to get better coffee—a prerequisite for doing good research. In my latter years, it's been truly enjoyable to work with a younger set of ascendant PhD students, in UCB-BAR and adjacent groups, including Arya Reais-Parsi (who provided valuable feedback on this dissertation), Kevin Laeufer, Vighnesh Iyer, Hasan Genc, Ben Korpan, and especially Jerry Zhao and Abe Gonzalez, whom I've previously mentioned for their contributions to FireSim. I leave with a strong sense the group is in good hands.

Faculty at UC Berkeley have played an important role in getting me to the finish line. I'd like to thank John Wawrzynek for being my first year advisor and for welcoming me warmly to Berkeley; Robert Leachman, who graciously served on my qualification and dissertation committees; William Kahan and Jim Demmel, for their mentorship on the EDP project; and Bora Nikolić, for being a pillar of the ASPIRE and ADEPT labs, and for having a better understanding of the problems FireSim should be solving long before I did.

Additionally I'd like to thank the staff of the ASPIRE and ADEPT Lab, and the EECS department at large. This includes the current and former administrative staff of these labs, notably Tami Chouteau, Ria Briggs and previously Roxana Infante, who have been consistently amazing; Kostadin Ilov, our ever-buoyant tech admin, who was always eager to tweak our FPGA cluster at my behest; and Shirley Salanio, who helped me through some of the more challenging times at Cal, and whose hard work made it possible for me to take Japanese as my outside minor. It remains one of the great mysteries of the EECS department how Shirley manages to find the time to be so supportive to so many graduate students.

My journey to graduate school at Cal was born out of relationships in undergrad. It begins with Rajiv Chopra, who really got me excited about becoming an academic in the first place. In my fourth year, I had the privilege to work with Jonathan Rose and his then PhD student, Alex Rodionov. Alex took me under his wing and laid the groundwork for how I think about designing digital systems. Jonathan gave me some of the best early advice about navigating graduate school; I wish I listened to him sooner and more carefully. I also need to thank many of the engineers I worked with at Altera (now part of Intel) before coming to Cal, notably Ivan So, Simon So, Byron Sinclair, Elias Ferzli, Ahmed Kammoona, Val Manohararajah, and most of all, the late Ivan Blunno, to whom I've dedicated this dissertation. Ivan encouraged me to go to graduate school even though I would've loved to hack on FPGA CAD tools under his supervision.

Circling back, there are three friends from UCB-BAR I must thank particularly:

- Jack Koenig. It was a pleasure to be frequent collaborators on some great (FASED, Golden Gate) and some not-so-great projects (EAR, EDP). But hey, 4288 bits to compute a double-precision dot product is a small price to pay to attend ARITH in

London (thanks Krste). Jack's drive to build awesome-and-usable tools is infectious, and something I tried to carry into the FireSim project.

- Andrew Waterman. While his technical reputation precedes him (see RISC-V, acknowledgments of other dissertations), Andrew was my graduate school champion: he believed in me even when I did not. I can confidently say I would have left in my third year if it were not for him. I miss our sojourns to Ramen Shop, our conversations over Japanese-brewed, industrial lagers and the hours "lost" to Halo thereafter.

- Albert "The Man" Magyar. I've had the pleasure of working with Albert at various points of graduate school including on EAR (the "gratuitous amounts of cache" project), but our close collaboration on Golden Gate defined the latter half of my graduate school experience. Golden Gate was built on spontaneous conversations in the Cosmic Cube and in afternoon voyages to Yali's cafe. It's unfortunate that COVID-19 robbed us of these interactions in our final year of graduate school—my research certainly suffered for it and, more importantly, I missed his reliably witty company.

While at times my research was nearly all-consuming, friendships outside of my research circles provided some critical balance in my life. I have many fond memories from early graduate school with friends made during visit days, namely George and my first roommate, Alyssa. Rock climbing kept me sane over the past seven years with Berkeley Ironworks serving as a vital escape. Shout out to my partners over the years: Peter, Pascal, Nick, Josh, and David. I must thank Simon, Claire, and particularly, Michael for putting together some wonderful backpacking trips. Finally, I'd also like to thank my latest roommate, Gabriel, for welcoming me into his life, getting me excited about baking, and for showing me the true versatility of cilantro.

When times were rough at Berkeley, I often looked to friends from Toronto including Matt, Denise, Judith, Connor, Mohamed, Marissa, Laurence, Cindy, and others. I'd like to call out a few in particular: Kevin, his time spent in SF was a wonderful period of graduate school for me, I miss having the excuse to visit the city; Zimu, for always checking in with me despite my online reticence, it was hard living on a separate coast from him; Richard, for providing some of my fondest memories of graduate school forged in trips to San Deigo and Scotland; and finally, Kelvin: my stalwart high school friend. I struggle to articulate how grateful I am he came to Cal for his Ph.D.

I conclude by acknowledging the two constants in my life. First, my family, namely Mum, Dad, and my sisters, Emma and Laura—it's been a long seven years and they've been behind me the whole way. And second, Fiona, who's reliably put up with my shenanigans over seven years of a long distance relationship, and, through some mysterious alchemy, can always coax a smile out of me. Next time I hope we can agree to get admitted to the same grad school.

Chapter 1

Introduction

Today, the semiconductor industry finds itself in the midst of a storm emerging from two colliding fronts[1]. The first of these, the hot front, is an exponential growth in the diversity and volume of computing applications. While AI, specifically machine learning, has captured the zeitgeist, advances in semiconductor process technologies have made it possible to embed computing in nearly all aspects of human life. This manifests at the extremes as deeply embedded systems, often running in highly energy-constrained environments, and cloud-hosted services, running in multi-megawatt-scale datacenters, all interconnected via an ever more capable internet. All of this—from the smallest embedded microcontrollers to the largest datacentered-oriented servers, and the networking hardware that connects them—runs on transistor technologies whose fundamental physics have not changed since the mid-twentieth century.

Indeed, the "cold" front of this storm is the inevitable slowing of transistor scaling trends that have long been the hallmark of the semiconductor industry. First, we lost Dennard scaling [31]: a regime in which a shrink in transistor channel length and voltage would produce a proportional reduction in the delay of a circuit built from them. Dennard scaling drove an era of exponential increase in computing performance wherein one could simply re-implement an existing design in the latest process technology to realize large speedups. Dennard scaling ended in the mid-aughts when it became difficult to further scale transistor voltages without losing the ability to shut them off [76], forcing power density to increase. This put a thermal limit on how fast one could clock a chip and led microprocessor manufacturers to abandon development of higher-frequency, deeply pipelined machines.

In the years since, advances in computing performance have ridden on the back of the industry's better-known scaling trend, Moore's Law [74]. Even if transistors were not getting faster, they were still shrinking. These extra transistors could be translated into larger caches and prediction structures, physically wider vector and SIMD instruction pipelines, and most notably, more processor cores. In this time, we also saw the rise of the general-purpose GPU (GPGPU), whose more parallel architecture scales naturally to use more transistors

[1]I borrow this metaphor from my advisor, Krste Asanović.

than a conventional microprocessor. GPGPUs have been a fundamental driver in the success of many modern machine-learning techniques [61]. Unfortunately, GPGPUs are far from truly general-purpose; the bulk of the world's applications still run on conventional microprocessors where improvements offered by more cores and other microarchitectural enhancements are bearing less fruit.

With no radically better general-purpose computer architecture (based in silicon or otherwise) waiting in the wings, there is broad agreement that the only means to deliver continued advances in computing performance and energy efficiency is with specialization. In academia, many researchers are building accelerators for specific application domains, like graph computing [46] and machine learning [25]. In industry, there is a growing number of companies designing their own silicon where previously they would have purchased offerings from existing players. Notable examples include Tesla [13], Amazon [9], Google [52], and more recently, Apple[2] with their M1 SoC [6].

What this "specialization" should look like is the defining question of this era of computer architecture research. However, the more narrow concern of this dissertation is how a particular microarchitecture should be implemented in silicon. While application-specific integrated circuits (ASICs), offer the best potential for realizing these improvements, the non-recurring engineering (NRE) cost of a new design is enormous and growing (Figure 1.1). As a result, reconfigurable logic devices like field-programmable gate arrays (FPGAs) have long filled low-volume niches where the NRE of custom silicon (i.e., an ASIC) cannot be effectively amortized. The need for lower cost, energy-efficient hardware has also driven a resurgance in structured ASICs [35]: these are FPGA-like devices whose field-programability has been removed. Structured ASICs attempt to close the performance and energy-efficiency gap between FPGAs and ASICs while saving up to 90 % of the NRE of an equivalent ASIC [106]. Nonetheless, we believe the performance and energy-efficiency costs [105] of using these intermediate technologies is large enough to justify a redoubled effort to make standard-cell-based ASIC design more economical.

The large NRE of developing an ASIC is in part historical. Years of advantageous scaling trends have bred a business model in which, at least in advanced technologies, relatively few unique designs are taped out in enormous volumes. Simultaneously, advances in general-purpose computers dissuaded investment in smaller-volume custom-silicon projects: why, after all, would one design an ASIC, when one could wait two years for the next Intel CPU? As a result, tools and methodologies have been designed and optimized around large NREs and large volumes to amortize them. Moreover, the established players in the electronic design automation (EDA) industry, responsible for developing the tools used to design ASICs, have little short-term incentive to alter this business model, because the sheer complexity of modern EDA tools makes it difficult to license to smaller players at lower costs. For the benefits of custom silicon to be attainable for the non-Apples and Googles of the world this will need to change.

[2]Apple has long been designing its own SoCs for its mobile phones, but used Intel SoCs in their laptops and desktops.

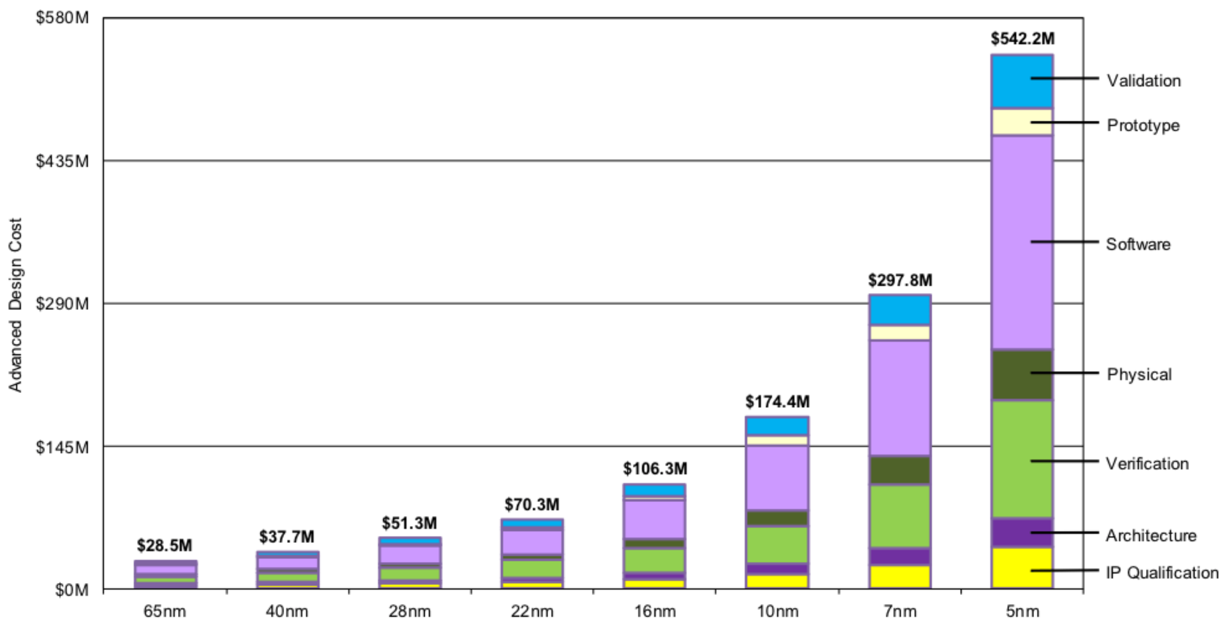


Figure 1.1: Early design non-recurring engineering (NRE) cost at various feature dimensions. Source: IBS.

Part of the problem is that there appears to be no silver bullet for reducing ASIC NRE, as there are many disparate sources that often span multiple domains of chip design (Figure 1.1). Instead, we will need many diverse tooling improvements united under a reimagined methodology for building custom silicon. At Berkeley, we have articulated one such vision inspired by agile software-engineering practices [68] and have built tools to support it. These tools include Chisel [12], a more productive hardware-design language (HDL) based in Scala, and FIRRTL [50], a flexible intermediate representation for hardware that makes it easier to build compilers. Perhaps the largest contribution to this vision has been the RISC-V instruction set architecture (ISA) [102]. Being open and free-to-use, RISC-V enables chip designers to build customized microprocessors while extending an open-source software toolchain. This avoids the costs and legal encumbrances of using a proprietary ISA, or the massive engineering burden of developing and supporting a fully custom ISA. RISC-V adoption has been meteoric in recent years, both in academia outside of Berkeley [82, 30, 87], and in industry, where a growing body of companies, notably SiFive, Western Digital, and Nvidia have been developing new RISC-V implementations.

One unaddressed deficiency in chip design that drives verification, validation and software development costs, is the lack of a good full-system simulation technology. Ideally, such a technology would be:

- **Fast.** As fast as a silicon prototype, so as to enable running full system stacks and complete applications.

- **Detailed.** It would simulate SoC's timing characteristics exactly.
- **Productive.** It would be easy to debug and fast to recompile. It would feel much like a software-based simulator.
- **Inexpensive.** It would be cheap to deploy so that it could be made widely accessible to hardware and software design teams and support running large parallel experiments.

In practice, existing simulation technologies are forced to prioritize two of these objectives. Software simulators, despite being easy to use and relatively inexpensive, are much too slow to run full-system simulations when providing a cycle-accurate model of the chip. For speed, chip designers are forced to turn to hardware acceleration, in the form of FPGA prototyping and hardware emulation. Of the two, FPGA prototypes tend to be faster and less expensive, and so see extensive use in software development and regression testing later in the design cycle. Conversely, hardware emulation is slower and considerably more expensive, but offers a software-simulator-like debugging experience that makes it a critical tool for pre-silicon verification. We will expand more on these differences in Chapter 2.

The central question of the simulation work underway at Berkeley asks if it is possible to build a simulation platform that can marry the speed of FPGA prototyping, and the productivity of hardware emulation, while being radically cheaper to deploy. Such a technology would put hardware emulation in the hands of smaller research groups, and make it more widely available in companies where scarce emulation resources must be carefully scheduled. Additionally, if it is fast enough, it could subsume the role of a conventional FPGA prototype, reducing the engineering burden of having parallel emulation and FPGA prototyping infrastructures. If this technology is going to be inexpensive, two things are clear. First, it must use commercial off-the-shelf (COTS) FPGA platforms, and not custom devices or system integration as is true of all commercial emulators. Second, this technology's software ecosystem, including tools and IP libraries, will need to be open-source with the hope that they can be developed and maintained by a community of users instead of a private team of engineers.

This is no small undertaking, but it is one made easier by recent technology trends. Firstly, modern FPGAs are enormous, and continue to be among the greatest beneficiaries of Moore's law since they can naturally scale to use larger transistor budgets. Advances in multi-die integration technologies have made it easier to reliably manufacture even larger FPGAs. This makes it possible to simulate considerably larger systems on a single FPGA. Secondly, FPGAs are now available in public cloud and provided by major vendors like Amazon Web Services [2]. This means a team can avoid the expense of building and maintaining their own local FPGA cluster, and can use newer FPGAs as they become available. Cloud services also provide an elastic supply of FPGAs, allowing users to scale out experiments as needed, instead of needing to provision a local cluster for the worst case. While the hourly rates of using cloud-based FPGAs are currently non-trivial, we expect these to fall in the future as the costs of these services are amortized. Finally, it would be difficult to build out a hardware emulation infrastructure without the aforementioned advances in open-source tooling. A compiler lies at the heart of all hardware emulation systems: FIRRTL and its Scala-based compiler infrastructure provide flexible abstractions for translating a design into an FPGA-hosted emulator. FPGA-hosted emulation infrastucture requires customized hardware, and more productive hardware-design languages, like Chisel, make it considerably easier for a small team to build it. Finally, for an open-source infrastructure to see use, it requires open-source input designs to both encourage adoption and inform development. Here, we can leverage a growing body of RISC-V based processors, such as Rocket [8] and BOOM [22], to feed into our system.

While it initially began as a project to study warehouse-scale computers, the FireSim [53] project has been our attempt at building out this new technology. FireSim has been the basis for a number of emulation-related research papers at Berkeley: we have studied detailed DRAM timing models [16], and non-invasive, full-system profiling techniques [54]. FireSim-adjacent projects using the same underlying compiler, MIDAS [56], have explored novel techniques for rapid debugging [57], and energy estimation [59, 58]. Over time, features from these projects have gradually been integrated in mainline FireSim, where they are employed by a growing user base.

Unfortunately, limitations with MIDAS limited its scope to simple designs. First, these designs were small: they could be directly implemented on a single Xilinx VU9P FPGA, a mid-range device from the Ultrascale+ family with approximately 1.18 million lookup tables (LUTs). To provide more context, prototyping-optimized FPGAs from the same family can have as many as 8.17 million LUTs (VU19), and large modern SoCs must be partitioned over them [4]. Second, these systems possessed only a single clock domain. Conversely, modern SoCs have dozens of clocks whose frequencies may change during execution. Without the ability to simulate even a small number of fixed-frequency clocks it was difficult to get truly accurate performance estimates with FireSim. Addressing these two challenges drove the design of a new optimizing compiler called Golden Gate [70].

To simulate larger systems, Golden Gate uses multi-cycle resource optimizations to replace FPGA-hostile blocks. Golden Gate automates many techniques deployed in prior academic work in FPGA-accelerated microarchitecture simulation to radically improve the capacity of single FPGA. These optimizations are the focus of Albert Magyar's dissertation [71]. Addressing the second challenge of simulating systems with more realistic clock organizations is the subject of this dissertation. The contributions of this work are threefold:

1. We describe Golden Gate, an optimizing compiler infratructure to deploy module-based multi-cycle resource optimizations (Chapter 4). This contribution is shared with Albert Magyar, and is also described in his dissertation [71].

2. A simple, FPGA-portable extension to Golden Gate to support systems with fixed frequency clocks that is compatible with multi-cycle resource optimizations (Chapter 5).

3. A general, distributed approach for simulating clocking structures, based off of prior work [23] in software parallel discrete-event simulation (Chapters 6 - 8). As above,

our implementation co-exists with all multi-cycle resource optimizations. This is the primary contribution of this dissertation.

All of the software described in this dissertation is open source. We make frequent reference to specific versions of the FireSim codebase to provide more context in each chapter. Code references will use `monospace typeface` where applicable. We hope this will help document features of FireSim as more users adopt it. At time of writing, Golden Gate and support for fixed-frequency clocks (Chapter 5) have been integrated into mainline FireSim and are under active use. The features described in Chapters 6 - 8 are tagged `pdes`, and will be merged in the future.

1.1 Previous Publication, Collaboration, and Funding

Portions of this work were published at the 2019 International Conference on Computer-Aided Design as *Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes*, though in this dissertation we focus on the design of the compiler. The specific resource optimization employed in that publication is described at length in Albert Magyar's dissertation [71]. Early results from our initial support for emulation of multiple fixed-frequency phase-aligned clocks, were published as *Accessible, Resource-Efficient FPGA-Accelerated Emulation Using FireSim* in a July 2021 IEEE MICRO special issue on FPGAs in Computing.

Donggyu Kim drove the bulk of MIDAS development based on his work on Strober [58]. In the "1.0" version of MIDAS presented in his dissertation [55], I contributed the design of the endpoint system. The final version of that compiler used by FireSim is described in Section 3.5.

The work uses features developed by other FireSim contributors. Notable contributors include Alon Amid, who developed much of the debug infrastructure, Howard Mao, who has contributed to nearly all subsystems in FireSim, and Sagar Karandikar, who ported MIDAS to use AWS and designed the cloud manager.

Finally, Golden Gate was the product of a tight collaboration between myself and Albert Magyar. Much of the complexity in our dissertations is tied to the FIRRTL-based software infrastructure required to analyze target SoCs and transform them into latency-insensitive networks. This common infrastructure is described at length in Chapter 4, and is the basis for the remainder of the dissertation.

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849, and NSF CCRI Award 2016662. Research was partially funded by ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate, and supported by gifts provided by Amazon Web Services and Xilinx. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Chapter 2

Full-System Simulation of SoCs

Full-system simulation is used ubiquitously in SoC design and verification. In this chapter, we commence with an exploration of the roles served by full-system simulation in the SoC design process, and discuss commonly used full-system simulation technologies with emphasis on hardware-accelerated forms like FPGA prototyping and hardware emulation. From there we pivot to study how computer architecture researchers employ full-system simulation for higher-level architectural and microarchitectural studies, and how academics have sought to use FPGAs to accelerate these simulations. We conclude by reviewing work done by the Berkeley Architecture Research (UCB-BAR) group over the past decade that attempts to build more cost-effective hardware emulators by applying techniques from academic FPGA-accelerated microarchitecture simulators as compiler transformations on SoC RTL.

To avoid confusion when speaking of computers simulating computers, we make a distinction between the *target*, the system being simulated, and the *host*, the system executing the simulation. The host is often not a single machine but a collection of interconnected machines, which may include CPUs, GPUs, and FPGAs.

2.1 A Tour of Full-System Simulation for SoC Design

Simulation is central to performing three fundamental tasks of SoC design.

1. **Prototyping:** "What thing should we build?" Prototyping serves as a means to rapidly evaluate different design points with an imperfect model of a proposed design.

2. **Verification:** "Did we build the thing right?" Verification serves to check, or prove, that a particular implementation correctly executes.

3. **Validation:** "Did we build the right thing?" Validation serves to show that the implementation fulfills the objectives set out for the system.

Both prototyping and verification can be applied at all levels of the design hierarchy. For example, given a specification of the system into which an accelerator is to be integrated, one could prototype different design points and verify an implementation of that accelerator. Validation, however, seeks to answer a system-level question that spans the entire computing stack. The surest way to validate a system is not in simulation, but at-speed with a physical prototype or the final product itself. However, waiting for a silicon prototype pushes validation late into the design cycle making it challenging or impossible to pivot the design of the system based on validation results. To perform *pre-silicon* validation, a fast and accurate full-system simulator is required. Here, SoC designers are confronted with a cost-fidelity-performance trade-off, and are forced to use multiple different simulation technologies at different points in this space.

2.1.1 CPU-Hosted Simulation for Prototyping

Architecture-level simulators such as QEMU [84], which model the system at the instruction-set-architecture (ISA) level and include a limited set of standard device models for I/O, are fast and inexpensive as they run on conventional CPUs. When augmented with simple timing models, they are ideal for doing initial system prototyping, as these models can be quickly modified and recompiled. However, as these timing models become more complex, they become more challenging to validate, and crucially, the throughput of these simulators rapidly declines.

Continuing in the direction of increasing fidelity, microarchitecture-level simulators, such as Gem5 [17] and MARSSx86 [78], are CPU-hosted simulators that provide configurable, "cycle-accurate" timing models of a complete system, including CPU pipelines, caches, and off-chip memory systems. These simulators can run target workloads at hundreds of kilo-instructions-per-second (KIPS), but are often much slower in practice when employing more detailed or custom models. This makes it practically impossible to run complete workloads, such as multi-threaded Java applications or SPECint2006 [47] with its reference inputs. Here, a common remedy is to employ statistical sampling techniques [107] to fast-forward to the region of interest on an architecture-level simulator, before executing O(100M) instructions at the desired fidelity.

While this approach has well-acknowledged shortcomings [45], it can be an appropriate vehicle for doing initial system prototyping. For radical proposals that involve aggressive microarchitectural changes or traverse multiple layers of the computing stack, this approach is often inadequate. This includes workloads that are multithreaded, or are long-running and irregular (such as managed-language workloads), for which it is difficult to collect meaningful samples without perturbing the system under evaluation [48].

2.1.2 CPU-Hosted Simulation for Verification

Since the aforementioned simulation techniques use abstract models of the target system instead of directly simulating its implementation, they are unproductive for system verification and validation once implementation begins. Here designers instead use CPU-hosted simulators that faithfully represent the implementation at a particular abstraction level.

For simulating digital components of the SoC, a register-transfer level (RTL) simulator, like Synopsys VCS or Verilator [97], is the tool of choice. Broadly speaking, RTL simulators do a cycle-by-cycle simulation of all target-state elements and the combinational functions that update them. Generally in these simulations the values on wires transition instantaneously and no delay through registers and combinational circuits is modeled. Supposing the underlying digital abstraction holds, RTL simulation is ideal for doing dynamic verification of a digital circuit. Small blocks compile in seconds, while complete SoCs can be compiled in ones to tens of minutes. RTL simulators are relatively easy to debug as they provide complete visibility over the state of the design over the entire duration of a simulation. The cost ($) of an RTL simulator comes mainly from software licensing fees, as simulators run on standard servers or desktop CPUs, though in many cases an open-source RTL simulator like Verilator can be used instead. Ultimately, the largest challenge in using CPU-hosted RTL simulators is that they have poor simulation throughput for large designs. Complete SoCs execute at hundreds to less than one Hz—much too slow to verify anything but short-running inputs (e.g., checking early system boot), or for full-system validation.

It's important to note at this point that higher-fidelity software simulation of the design is commonly used after the SoC is synthesized and implemented in a particular process technology. These simulations generally include more detailed models of analog components of the system (e.g, for PLLs), as well as combinational, wire, and parasitic delays that can only be accurately determined once the SoC has passed through physical design. These implementation-dependent delays can be back-annotated onto a gate-level netlist and then simulated to detect race conditions and other bugs that static timing analysis tools may not find. While back-annotated gate-level simulation is critical, notably for verifying an SoC's reset sequence, the additional fidelity only exacerbates the throughput limitation of CPU-based simulation.

For effective full-system validation and dynamic verification, considerably faster simulation throughput is required. While there are many techniques that can improve throughput on CPUs, such as multithreading and relaxing or restricting the timing semantics of the design language, the abundant fine-grained, often bit-level, parallelism of RTL simulation cannot be exploited by multiprocessors sufficiently to overcome the enormous slow down over a silicon prototype.

2.1.3 FPGA Prototyping

To build simulators that execute at rates closer to a silicon prototype, designers turned to fine-grained parallel hardware. One of the earliest forms of hardware-accelerated full-system evaluation emerged in the 1980s and used programmable logic devices, specifically FPGAs, to directly implement the design. This is known as *FPGA prototyping.*

Modern FPGA prototypes directly implement the SoC on one or more FPGAs, often with a custom board design that may include peripherals identical to those that would be deployed in the final system. FPGA prototypes are fast: small prototypes that fit in a single FPGA execute at tens to hundreds of MHz, while larger prototypes, which must be partitioned across multiple FPGAs, simulate at hundreds of kHz [88, 101]. Modern vendor-provided FPGA prototyping platforms, such as Cadence's Protium[19] platform, can run at 1-10 MHz by carefully managing inter-FPGA interconnect, and cleverly partitioning the design. Often, FPGA prototypes are inexpensive enough that they be can readily duplicated and distributed across hardware and software engineering teams. Relative to software simulation, prototypes have greater fixed costs (to buy, design, and/or license the prototyping platform), but more critically, are difficult to use [4]:

1. Poor design visibility makes it difficult to debug failing systems. Users must instantiate FPGA-specific debugging hardware which provides only a limited set of signals for a small window of time, as these tools consume considerable FPGA resources. If the bug is not found on the first iteration, the process must be repeated: the design must be resynthesized with a new set of sampled signals.

2. Long compile times (1s - 10s hours) make it difficult to iterate about a design point and lengthens the aforementioned debug cycle.

3. Large designs do not fit on a single FPGA and must be partitioned across multiple FPGAs either manually or with specialized tools. This makes the prototype hardware more expensive and decreases simulation throughput.

4. Many ASIC structures, such as multi-ported RAMs and clock generators, cannot be synthesized into FPGA fabric so must be replaced with an FPGA equivalent.

5. FPGA-specific I/O models are required to build out a complete system and using hardened IP on the FPGA may not be a good model of the target system. Performing software co-simulation of IO often reduces simulation throughput.

6. Prototypes are not natively deterministic making it difficult to reproduce certain classes of system failure, especially those that involve I/O.

7. Prototypes require a complete RTL implementation of the design.

These limitations make FPGA prototypes unproductive for full-system verification. However, in most cases FPGA prototypes are the fastest available full-system RTL model of the target and so are used extensively for pre-silicon software development and regression testing.

2.1.4 Hardware Emulation

Hardware emulation aims to provide productive full-system verification by marrying the speed of FPGA prototypes with the usability of software simulators. Hardware emulators tend to be expensive to license and run—millions of dollars per unit per year—making them a precious commodity that must be carefully shared across a company.

Each of the three major CAD vendors offer hardware emulation solutions. Mentor Veloce [90] and Cadence Palladium [20] are the historical market leaders in hardware emulation with the two jockeying for position with each update to their emulation platforms. Synopsys ZeBu [93] rounds out the offerings. Differences in the implementations of these three emulators put them at different points in the cost-usability-performance space. We describe their three implementation strategies in the following sections.

ASICs - Logic Processor Arrays

Perhaps the earliest form of hardware emulation traces back to IBM Research's Yorktown Simulation Engine (YSE [32]) and its industrial-strength successor project the Engineering Verification Engine (EVE [15]). These machines consisted of an array of small processors. Processors were specialized either for simulating logic (logic processors) or memory (array processors), and were interconnected with high-radix crossbars (256 x 256 in early machines). A compiler [62] would partition, map, and schedule the target onto this array. Both EVE and YSE could perform gate-level simulation in zero-delay or unit-delay (every gate propagation incurs a constant delay) modes. Cyclist [11] is a recent academic work that further explores this approach: it uses a homogeneous array of custom RISC-V cores attached in a 2-D mesh network.

Cadence's Palladium [20] emulators derive from EVE (which was for a time sold by QuickTurn Design Systems under the name CoBALT [34], before they were acquired by Cadence). Relative to competing emulation solutions, Palladiums have lower capacity per unit, but have the fastest compile times. Palladiums have the highest power consumption and, unlike the competing offerings, must be water-cooled.

Custom FPGAs

While EVE-like processor arrays can provide radically improved compilation speeds, since the compilation problem is fundamentally simpler than running FPGA place and route, one natural alternative is to modify conventional FPGAs for emulation. This might involve adding dedicated hardware to improve debuggability (e.g., to make it possible to capture full-visibility waveforms), adding I/O multiplexing hardware to ease the FPGA-partitioning problem (e.g., a hardware implementation of Virtual Wires [10]), and modifying the programmable fabric to better match the resources and interconnect required by ASIC designs. This is the approach taken by Mentor Graphics's Veloce [90] emulation platforms. Veloce emulators appear to have better capacity than Palladium, but slower compile times. Veloce emulators use less power and are air-cooled.

Commercial-Off-The-Shelf FPGAs

To some degree, hardware emulation tends to have large total-cost-of-ownership (TCO) because the leading emulators are both power-hungry and use custom silicon. To save some of that cost, a third alternative that sees industrial use is to use large, commercial-off-the-shelf (COTS) FPGAs but rely on custom tooling and system packaging to improve usability. Synopsys's ZeBu [93] platform does precisely this and leverages the largest available Xilinx

FPGAs (at the time of its release these were Virtex Ultrascale VU440s [93]). At time of writing, ZeBu provides the highest simulation throughput and the lowest power consumption but has a reduced debug feature set relative to other emulators. In its use of COTS FPGAs, Synopsys ZeBu is most similar to the work presented herein.

2.2 Hardware-Accelerated Microarchitecture Simulation

To build more cost-effective hardware emulators, this work draws from technologies developed in academia to accelerate cycle-level microarchitecture simulation (this is described see Section 2.1.1).

The first, most obvious way to accelerate these is to parallelize them over multiprocessors or networks of workstations. One early example of this is the Wisconsin Wind Tunnel (WWT) [86], which relied on a window-based, parallel discrete-event simulation engine to manage synchronization across workstations. For CPU-hosted simulators like the WWT and recent works like Graphite [73] to achieve good performance, they must reduce the synchronization overhead between partitions of the design by either modeling the target more abstractly, or by introducing additional target latency between partitions. While this may be appropriate for building coarser models, it is ineffective for accelerating RTL simulations and thus for building hardware emulators which require at least per-cycle synchronization.

So to build simulators that were both fast and cycle-accurate many academics turned to FPGAs, which by the 1990s and early 2000s had proven themselves as effective vehicles for ASIC prototyping and emulation. However, instead of directly implementing an ASIC design, here FPGAs would be used as a host for microarchitectural models written in RTL. While an early example of this approach is the Rapid Processor Emulator [14, 77], the bulk of the research in this domain can be attributed to the RAMP project [103].

2.2.1 Research Accelerator For Multiple Processors (RAMP)

The RAMP project began in 2005 driven by the realization that advances in computing performance would require exploiting other forms of parallelism beyond ILP, as the end of Dennard scaling would necessarily make single-threaded performance improvements more difficult to attain. The RAMP project's goal was to develop a shared full-system simulation infrastructure which would be better suited than traditional software simulators to study future thread-parallel machines. Member universities included UT Austin; CMU; UC Berkeley; University of Washington; Stanford; and MIT.

At the onset of the project three monolithic prototypes were built, each was designed to model a different class of target. RAMP Red, later known as ATLAS (Stanford) [104] was designed to study transactional-memory-based chip-multiprocessors, and supported up to 8 PowerPC cores. RAMP Blue (UC Berkeley) [60] was tailored for large-scale distributed-memory message-passing machines and used Xilinx Microblaze softcores to prototype the target system. Partitioned over 21 Berkeley Emulation Engine 2 boards (BEE2) [24], RAMP-Blue could simulate a system with as many as 1008 cores. Finally, RAMP White (UT Austin) [5] modeled cache-coherent shared-memory processors. It supported both PowerPC (when using a Xilinx host with a hardened PowerPC 405 core) and 32-bit SPARCV8 (soft core) targets. Each of these prototypes used the same host platform (BEE2), and were initially constructed using shared libraries and a common specification language called the RAMP design language [39].

Other, mostly later, projects tied to RAMP abandoned the shared infrastructure and explored different simulator design styles. ProtoFlex (CMU) [28] was an architecture-level simulator that demonstrated 16-way host-multithreading of a single FPGA-hosted functional model. ProtoFlex could switch between FPGA-hosted and CPU-hosted modes via a process it called transplantation. FAST (UT Austin) [26] was a cycle-accurate x86 simulator which leveraged a split, CPU-hosted functional model and FPGA-hosted timing model. RAMP-Gold (UC Berkeley) [95] used FPGA-hosted timing and functional models with 64-way host-multithreading to realize a larger target on a single FPGA. To model a datacenter-scale target, DIABLO (UC Berkeley) [94] stitched together 24 instances of a modified version RAMPGold to simulate 3072 interconnected servers. Finally, HASim (MIT) [79] also used FPGA-hosted timing and functional models, but provided more detailed pipeline and memory hierarchy models. Later work studied partitioning HASim over multiple FPGAs [36] and showed that by using two FPGAs HASim could host eight times as many cores, due to improved resource sharing between virtual instances.

Around the same period, other groups not associated with the RAMP project explored using FPGAs for microarchitecture simulation. One notable example is DART [100], an FPGA-based Network-on-Chip (NoC) simulator that used multithreading, like many RAMP simulators, but leveraged NoC-specific model abstractions to permit a wide range of runtime-reconfigurable model parameters.

2.2.2 FAME Taxonomy

One output of the RAMP project was the FPGA Architecture Model Execution (FAME) [96] taxonomy of FPGA-accelerated simulation work, which distills many of the contributions of the works above into three dimensions: host decoupling (FAME-1), abstract RTL (FAME-2), and multithreading (FAME-4). Like the RAID classification, each of these dimensions can be composed, in this case to produce FAME-0 through FAME-7 simulators. For example, simulators employing multithreading tend to be host decoupled—under this taxonomy, these would be referred to as FAME-5 simulators. Similarly, RAMPGold [95] and HASim [79] are FAME-7 simulators: they deploy all three techniques.

2.2.3 FAME-1: Host Decoupling

In host-decoupled FPGA simulators, a target cycle of simulation executes over multiple FPGA-host cycles. In contrast, a conventional FPGA prototype executes a single target-cycle on every FPGA-host cycle; multiple clocks can be prototyped using multiple host clocks with the same relative frequency relationship as exists in the target. With host decoupling, ASIC structures that map inefficiently to FPGA fabric may be replaced with optimized-for-FPGA structures that take more host cycles to execute, but save FPGA resources and improve host-cycle time. One classic optimization replaces multi-ported register files and CAMs with a dual-ported BRAMs accessed over multiple cycles. Additionally, host-decoupling permits the simulator to tolerate variable latencies in the host without sacrificing simulator performance or changing the target-time behavior of the simulation. Nearly all academic FPGA-accelerated simulators employ host decoupling. Unlike FPGA prototypes, commercial FPGA-based emulators are necessarily host decoupled to make them execute deterministically and to support a wealth of additional features that may require halting parts of the emulation. We expand on mechanisms for implementing host decoupling in Chapter 3.

2.2.4 FAME-2: Abstract RTL

In an abstract-RTL FPGA-accelerated simulator, components of the simulator do not model the implementation RTL exactly. Abstraction permits simplifying components of the target, trading simulation fidelity for FPGA-resource savings. Additionally, abstract models can be made reconfigurable in ways the implementation RTL cannot (e.g., a latency-pipe model can expose its latency as a runtime-programmable register). In practice, the FAME-2 dimension of the taxonomy represents a spectrum. Most academic FPGA-accelerated simulators are either partially or completely abstract-RTL simulators. That said, even commercial hardware emulators are to some extent abstract as generally some ASIC features, like clock generators and other analog blocks, may need to be replaced with an equivalent model provided by the emulation platform.

2.2.5 FAME-4: Multithreading

In a multithreaded FPGA-accelerated simulator, like HASim or RAMPGold, multiple virtual instances of a block or module within the target are simulated using a single physical datapath on the FPGA. The target state is duplicated once per virtual instance, and a scheduler selects which virtual instance should be simulated in a given host cycle. ASIC logic tends to be expensive when mapped to FPGA fabric; in FPGA prototypes, designs tend to be logic (LUT) constrained, which leaves much of the FPGA's embedded BRAM unused. Multithreading improves the mapping efficiency of the target, by reusing the expensive logic over multiple copies of target state which can be mapped into abundant FPGA BRAMs and registers. To the best of our knowledge, commercial hardware emulators and FPGA prototypes do not use multithreading schemes for sub-components of the target.

2.2.6 RAMP Retrospective

Ultimately, RAMP-style FPGA-accelerated simulation failed to take off in the computer architecture community. There are a number of technical explanations for this, including the lack of a good open ISA: only SPARC was available at the time and this in part motivated the design of RISC-V. From a simulator-implementation perspective, many challenges RAMP faced derive from aforementioned difficulties with using FPGAs to prototype ASICs (see Section 2.1.3). Three more simulation-specific challenges include:

- **Large Relative Barrier to Entry.** Open-source software simulators can run on machines researchers already possess, whereas FPGA-based simulators require expensive hardware. For instance, the early RAMP prototypes all used the BEE2 board. While using smaller, more inexpensive boards would reduce initial capital costs, it would do so at the expense of reduced simulation capacity. Simulators using smaller hosts, like RAMPGold, required extensive resource-optimizations to support research-worthy target designs.

- **Reproducibility of Results.** Since RAMP simulators were tied to particular FPGA host platforms, other researchers would need to purchase the same host to reproduce published results. Even if researchers had access to their own FPGAs, they generally could not run a simulator designed for a different host on their own FPGAs without modification to simulator.

- **Modeling Complexity.** Designers of RAMP simulators have said that designing models was more difficult than writing RTL for the matching implementation. Consider a processor pipeline: to model detailed "cycle-accurate" behavior of that pipeline the model designer must write RTL that contains much of the same complexity inherent to the actual pipeline's design in order to properly capture the all hazards that may affect the processor's performance. Furthermore, designers must add still more complexity to support modeling a space of different processor designs, either at compile time by generating different model RTL, or at runtime, by adding logic to permit reconfiguring the simulator. Finally, designers must also apply multi-cycle resource optimizations to support simulating machines of reasonable scale, which adds still more complexity. Taken together, this complexity makes the model more difficult to debug than an already difficult-to-debug FPGA prototype, and to add insult to injury, these models, like their software counterparts, must still be validated against a real implementation.

2.3 Recent Work at Berkeley

At the end of the RAMP project it was clear that writing performance models for FPGAs was not going to be tractable without major breakthroughs in improving FPGA usability. Since evaluating an RTL implementation with an EDA flow is still required to meaningfully assess a design's quality-of-result (QoR), notably its critical path delay, effort spent writing FPGA performance models would be better spent writing the implementation. If the RTL design process could be made more productive, computer architecture researchers may be more willing to conduct microarchitecture studies using realizable RTL designs instead of software simulators. In this model, once an RTL implementation is ready it could be transformed into a RAMP-like simulator using a *FAME compiler*. This circumvents the aforementioned model complexity challenge, as these simulators would essentially be area-optimized hardware emulators of the design: they would exactly represent the input design's behavior and so would not need to be validated against silicon.

This was one vision that drove research done at UCB-BAR over the past decade. To address the RTL design productivity challenge, we developed Chisel [12]. Chisel, an RTL design language hosted in Scala, allows designers to specify rich hardware generators by leaning on the host language to provide powerful metaprogramming features not available in SystemVerilog or VHDL. Instead of performing parameter sweeps in a software model, with a generator the user explores the design space by elaborating and evaluating different *instances* of the design produced by different generator configurations. The Rocket Chip SoC generator [8], now maintained by SiFive, is one example of this: it generates complete RISC-V SoCs including core pipelines, private caches, uncores, and common periphery devices. Rocket Chip lets the user stitch together near-arbitrary networks of disparate devices using its Diplomacy library [29], developed by SiFive, which leverages Scala's type system to provide intelligent, area-optimized system integration. The Berkeley Out-of-Order Machine (BOOM) [22] and Hwacha vector-fetch processor [66] generators provide additional core IP that can be integrated into a Rocket Chip SoC.

To support writing reusable compiler transformations on generated instances, in Chisel3, Chisel2's internal representation (IR) of an RTL circuit and its lowering transforms were replaced with FIRRTL (Flexible Intermediate Representation For RTL) [50] and its Scala-based compiler. During elaboration, Chisel3 emits a high-level form of FIRRTL. The FIRRTL compiler then lowers the instance to Verilog while scheduling user-provided passes. These passes can be used to tailor an instance to a particular backend: for instance, when preparing a design for an ASIC implementation FIRRTL memories can be replaced by black boxes corresponding to technology-specific SRAMs. Alternatively, if the same design is destined to become an FPGA prototype, these memories could be implemented using double-pumped FPGA BRAMs to save FPGA resources.

2.3.1 Strober and MIDAS

The first FAME-compiler-like work done at UCB-BAR can be found in the Strober [58] energy modeling project. While gate-level simulation-driven power estimation using commercial tools like Synopsys PrimeTime PX provides accurate pre-silicon results, gate-level simulation runs much too slowly to conduct system-level microarchitectural studies using complete workloads. Instead, Strober generates an accurate average power estimate by sampling the execution of the workload running on a fast, FPGA-based simulator and then replaying those samples for short durations in gate-level simulation. By selecting an appropriate number of samples, average power dissipation for the SoC can be estimated within a desired error bound.

To realize this, Strober needed to automatically generate an FPGA-based simulator from ASIC RTL with the ability to capture complete RTL state snapshots and IO traces. While these features were available in commercial emulation tools, no reusable open-source flow existed at the time. Since Strober predated FIRRTL, it modified the Chisel2 backend to:

1. Host-decouple input RTL to support halting the simulator to capture state snapshots. This is called a *FAME-1 transformation*.

2. Inject a shadow scan-chain to read out target state.

3. Elaborate a simulation wrapper around the transformed target. This wrapper included IO channels that can buffer a limited IO trace, a latency-pipe timing model for the target's DRAM memory system, as well as a simulation control bus.

Strober simulators are co-hosted by an FPGA and a CPU – a *driver* process writes to memory-mapped registers accessible on the simulation control bus to advance the simulator and to initiate state snapshot capture.

Initially independent of Strober, MIDAS began as class project to build an FPGA-based performance simulation framework. It had its own FAME-1 transform and a nascent library of more detailed memory-system models (this would later become FASED [16]). Seeing an opportunity to reuse work, we merged these two projects. MIDAS, as presented at CARRV2018 [56], retained all of Strober's features but used Chisel3 and FIRRTL. Crucially, MIDAS vastly improved simulation performance and made it easier to support co-simulation of other I/O models.

MIDAS would become the basis for a number of research projects at UCB-BAR. To improve simulator debuggability, DESSERT [57] leveraged MIDAS's state snapshotting features to perform *ganged simulation*. Here one instance of a simulator runs ahead of a second lagging instance, to detect a simulation error. A detectable error could be either a fired hardware assertion, which had been synthesized from the source FIRRTL, or a commit-log mismatch between the target processor and an online golden model. On detection of an error, the leading instance instructs the lagging instance to capture a state snapshot of the target before the error occurs. Using the replay feature, a full visibility waveform of the failing target design could then be generated. While commercial emulators provide rich sets of state-snapshotting features, DESSERT's differs in that uses two simulators to let the simulation advance at full-throughput to the point of failure. Simmani [59] revisited power estimation but, instead of using complete state snapshots, injected a statistically selected set of performance counters into design which can be used to provide a dynamic estimate of power dissipation. Finally, MIDAS's DRAM timing models were published as FASED [16].

2.3.2 FireSim

As Strober was being developed, UCB-BAR continued to study new architectures for datacenters, notably as part of the FireBox [7] project. FireBox was an early example of a disaggregated datacenter and relied on high-bandwidth photonic networking made possible with relatively inexpensive silicon-integrated photonics, a focus of earlier work done by the lab. To simulate FireBox, we built FireSim [53]. Building on DIABLO, FireSim interconnected MIDAS-generated simulators with a distributed, CPU-hosted network model to build a cycle-accurate, warehouse-scale computer simulation. To overcome the usability challenges of DIABLO, FireSim relies extensively on automation. Instead of using a custom host platform, FireSim uses the public cloud, specifically Amazon Web Service's Elastic Computer Cluster (EC2). When the user wishes to run a simulation, FireSim's *manager* program requests as many FPGAs nodes (to host MIDAS-generated simulators) and compute nodes (to host switch models) as required. The manager's ability to dynamically spin-up and tear-down simulators deployed to EC2 makes it easy to coordinate simulations of near-arbitrary size. Armed with this flexibility, in our ISCA2018 publication [53] we were able to reproduce behaviors observable in real datacenters at a variety of different simulation scales.

Since the 2018 ISCA publication, FireSim has evolved to become a general-purpose hardware emulation environment for Rocket Chip-based SoCs. By optionally removing the network simulation, the manager can instead batch out parallel workloads to multiple independent simulator instances, making it productive for doing performance evaluations of SoC-scale systems. In addition to the manager, FireSim provides:

1. A FAME compiler to generate an emulator from FIRRTL. Initially this was MIDAS, but as of version 1.6.0 it has been replaced with Golden Gate [70].

2. Device libraries for modeling periphery devices commonly integrated into Rocket Chip SoCs. This includes a block device, UART, and a tether model in addition to a NIC model used to perform networked simulation.

3. FireMarshal [81], a utility to automate building Linux distributions for running on RISC-V based platforms.

4. An example Rocket Chip-derived generator, FireChip, which can instantiate many of the most commonly used RTL libraries in the Rocket-Chip ecosystem. FireChip serves as a useful starting point for a new user wishing to build their own generator.

FireSim has been used both in academia and industry, primarily for doing performance evaluations of new microarchitectural features implemented as extensions to Rocket Chip. At Berkeley, FireSim was the basis for FirePerf [54], which introduced improved instruction tracing, and non-invasive performance counter integration to perform rapid hardware-software co-design of the NIC and the Linux networking stack. FirePerf features have since been integrated into mainline FireSim. Academically, FireSim has been used for microarchitectural performance evaluations, and commercially, it has seen use at Intensivate, Esperanto, and

SiFive. FireSim is also being used as a RISC-V platform in DARPA's FETT [1] bug bounty program [42].

As a final note, in 2020 UCB-BAR released Chipyard [3], which unifies many of the aforementioned SoC design tools and IP, including many open-source libraries developed by SiFive, under a single environment. In Chipyard, FireSim is a library for doing hardware emulation; many standalone utilities that were previously hosted in FireSim but aren't strictly related to hardware emulation support, such as FireMarshal, have been hoisted up into Chipyard.

2.4 Motivations for Golden Gate (MIDAS II)

As adoption of FireSim started to increase, it was clear to us that limitations with its FAME compiler, MIDAS, prevented it from seeing more widespread use. Firstly, MIDAS's existing FAME-1 transform supported systems with only a single, fixed-frequency clock domain, and barred use of asynchronous resets. This precludes simulating most realistic systems and made it challenging to validate FireSim against existing RISC-V silicon. Secondly, capacity challenges with EC2 FPGAs made it difficult to simulate larger systems, notably the test-chips UCB-BAR was designing concurrently. Without the ability to do multi-FPGA partitioning, and without any automated application of RAMP-style optimizations, FireSim was limited to supporting only relatively small SoCs. Taken together, these restrictions kneecapped FireSim to being a limited, albeit open-source, hardware emulation environment for primitive Rocket Chip-based SoC designs.

To address these two concerns, we set about designing a new FAME compiler called Golden Gate [70]. Before we describe its initial implementation, in the next chapter we review different target formalisms and implementation strategies for building FPGA-hosted, discrete-event simulators.

Chapter 3

On The Design of FPGA-Based Discrete-Event Simulators

For the purpose of this dissertation, it is insightful to classify FPGA-based RTL simulators along two dimensions: timekeeping strategy and control granularity. For simplicity, here we consider only single-FPGA hosts, but we note that this discussion can trivially extended to multi-FPGA hosts. This chapter was influenced by discussion in Pellauer and Vijayaraghavan et al. in *A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs* [80].

Timekeeping strategy refers to how the simulator tracks simulated time (target time). *Explicit timekeeping* (ET) simulators having dedicated state to track time (in seconds) whereas simulators with *implicit timekeeping* (IT) instead rely on target-cycle count as a proxy. Implicit timekeeping represents an implementation optimization as additional FPGA resources required to track and manage timestamps are unneeded. For this optimization to apply broadly across the simulator, in general, target clocks must have fixed frequency and phase, and all simulation events must be synchronous to these clocks.

Control granularity refers to how time is permitted to advance across the simulator. At one extreme there exist simulators with *centralized control* (CC). These track time in a single location or are globally statically scheduled (i.e., the entire simulator advances in lock-step from timestep to timestep). Conversely, there are simulators with *distributed control* (DC). These simulators are parallel systems that can be represented as a directed graph of *logical processes* (LP). Logical processes, the nodes of the graph, communicate by sending *messages* over the graph's edges. Each LP locally tracks simulation time and can advance independently: they are themselves smaller discrete-event simulators. DC simulators can be coarse-grained, with LPs simulating core-scale components, or fine-grained, where LPs model blocks on the scale of CAMs, RAMs, or combinational circuits, like multiplexers.

Taking the product of these two dimensions produces four classes of hardware-accelerated RTL simulator:

1. **Implicit Timekeeping, Centralized Control (ITCC)**: FabScalar FPGA mod-

els [33].

2. **Implicit Timekeeping, Distributed Control (ITDC)**: RAMP simulators [95, 79], MIDAS-generated simulators [53].

3. **Explicit Timekeeping, Centralized Control (ETCC)**: Yorktown Simulation Engine [62], modern commercial emulators[1].

4. **Explicit Timekeeping, Distributed Control (ETDC)**: multi-chip commercial emulators[2], and this work.

For all intents and purposes, FPGA-based DC simulators are FPGA-hosted, parallel discrete-event simulators. Parallel, discrete-event simulation (PDES) has been a vibrant area of research since the 1980s, but PDES researchers have focused nearly exclusively on non-FPGA hosts (including multiprocessors, GPUs, supercomputers, and networks thereof). In the next sections we briefly introduce relevant PDES work, and explain how prior work in that field translates to hosting DC RTL simulators on FPGAs. First, we explore how to define simulation performance, and then study some CC simulator designs to help motivate the use of DC simulators despite their increased complexity.

3.1 An Iron Law For FPGA-Based Simulator Performance

If SoC designers turn to hardware emulation for speed, it is critical to understand how simulator design affects simulation throughput. For IT simulators with a single target-clock domain, Pellauer et al. [80] present a simple performance equation which, like the iron-law of processor performance that inspired it, breaks down the performance of a complete simulation into a product of terms:

$$f_{sim} = \frac{cycles_t}{cycles_h} f_{fpga} \tag{3.1}$$

Where,

$$\begin{aligned} f_{sim} &= \text{throughput of the simulator (target-cycles per second, Hz)} \\ f_{fpga} &= \text{the clock frequency of the host FPGA (Hz)} \\ cycles_h &= \text{the total number of host cycles over which the simulation executed} \\ cycles_t &= \text{the total number of target cycles simulated} \end{aligned}$$

[1]As their implementations are proprietary, we can not say so definitively.

[2]As above.

The right term of this equation, host frequency (f_{fpga}), is set by the critical path delay of the simulator. Depending on the simulator's design and the target it simulates, this may correspond to a critical path in the target design or a path that drives scheduling logic in the simulator itself [80] (i.e., some part of the circuit deciding whether to advance forward in simulation time). The left term, a ratio of host to target cycles, is a measure of the microarchitectural efficiency of the simulator. In an FPGA prototype, this term is effectively one: every host clock cycle simulates a target clock cycle. In a host-decoupled simulator, in practice, it is always less than one. We note that while it is possible to simulate multiple target cycles per host cycle by unrolling successive cycles of execution, there is no incentive to do so globally across a simulator, as it would double resource utilization, and likely double the critical path delay of the simulator. In DC simulators of target machines with multiple clock domains, it may make sense to do this for relatively small LPs in the fastest clock domains, if the simulator is rate-limited on those LPs. Given this, Pellauer et al. tend to the more-intuitive reciprocal of this term and dubbing it the *FPGA-cycles-to-Model-cycles Ratio* (FMR). We give it in Equation 3.1. Unless it can be deduced statically, FMR is only usefully defined over a non-trivial interval of simulation.

$$FMR = \frac{cycles_h}{cycles_t} \tag{3.2}$$

As an example, consider a five-read, three-write register-file modeled by an LP that uses a dual-ported BRAM. If the LP statically schedules all eight accesses, it will have an FMR of four. Conversely, a more clever design that dynamically schedules accesses only for ports that are used could have lower FMR (and could approach unity). Continuing with this example, one could gang together multiple BRAMs or LUTRAMs to build a more highly ported RAM structure capable of more port accesses per cycle [65, 64]. While this could reduce the FMR of the statically scheduled implementation from four to as low as one, it comes at the expense of FPGA resources and, potentially, a longer simulator critical path which may undermine the FMR improvement.

To apply Equation 3.1 to DC simulators with a single target clock domain, we can modify it to use the FMR of the LP that has executed the fewest number of target cycles. However, when LPs may advance only a bounded number of cycles ahead of slower LPs in the system, for a sufficiently long simulation Equation 3.1 above returns approximately the same result for all LPs.

3.2 Centrally Controlled (CC) Simulators

When first building a simulator, it is natural to want to coordinate time globally across the FPGA. It is simpler to implement, there are fewer potential sources of causality errors, and it easier to capture snapshots of the simulated system at particular points in time. In a system where different components of the target may take differing, or dynamically varying, number of host cycles to execute, there are two common approaches. The first is what we call *static-barrier synchronization*. Note, Pellauer et al. [80]. instead use the term *unit-delay simulation*, but this differs from how the term is used in other simulation contexts, where it applies how delays in the target are modeled (see the YSE [32] for an example). Here the simulator is granted a fixed N host-cycle budget to compute each target cycle, where N is equal the largest possible latency a sub-block may take to execute (i.e., in isolation its worst-case FMR is N). While simple to implement, this will result in wasted host cycles as in many target cycles the full N cycle allocation may be unneeded. We also note that while it is relatively easy to provide tight bounds for models of many on-chip blocks, like multi-ported RAMs, it is more difficult to do so for I/O devices, which may require large N in the worst case, but far less on average.

Instead of selecting N statically, a natural optimization one could make is to allow sub-blocks to signal when they have finished computing. Here, a centralized controller aggregates done-signals from all blocks in the system and steps the simulator only once all have been asserted (a wide AND-reduction). This is called *dynamic-barrier synchronization*. While this removes wasted host idle-cycles (FMR can fall below N), it introduces new control signals that often set the critical path delay of the simulator: these signals must be routed to a central location on the FPGA, propagated through a potentially wide AND-reduction network, and then fanned out again across the FPGA. This problem is exacerbated in modern FPGAs, like the VU9P devices used in Amazon's EC2 F1 instances, which are composed of multiple dies[3], as these control signals must use relatively scarcer and higher-delay inter-die interconnect. Additionally, the scarcity of inter-die interconnect puts increased pressure on the router, and for high utilizations often results in unrouted nets and other related DRC errors. To improve simulator frequency, it is natural, perhaps even necessary, to pipeline these signals at the expense of a fixed increase to FMR: unless simulation of target cycle can be overlapped, FMR will increase by one for each additional pipeline stage. To make things worse, while FPGAs have scaled in capacity, their logic delays, and thus achievable f_{fpga}, have seen little improvement. Since an increase in capacity begets more sub-blocks, the larger AND-reduction network is not offset by logic-delay improvements which further lengthens the critical path of the simulator.

It is possible to hybridize these two approaches as demonstrated by Dwiel et. al. [33], where they introduce a notion of *minimum FMR* (MFMR). In their work they took Fab-Scalar [27] out-of-order microprocessors and synthesized them onto FPGAs: their elaboration flow emits multi-cycle models for CAMs and RAMs whose microarchitecture guarantees they will complete within the user-specified MFMR. The centralized controller need only be aware of longer-latency events that may exceed this MFMR (specifically instruction and data cache misses, and complex arithmetic operations [33]) reducing the size of the AND-reduction network.

[3]In Xilinx parlance, each logic die is referred to as a super-logic region (SLR).

3.3 Distributed Control (DC) Simulators

These trends in FPGA scaling make it necessary to decentralize timekeeping to build fast simulators. Having more LPs, each of which manage fewer control signals that are routed locally, improves f_{max} while easing the routing and DRC challenges of centralizing control. The caveat is that distributing control can substantially increase simulator design complexity. Since LPs can decouple in target time, causality errors can arise if LPs act on messages out of temporal order. Conversely, if LPs are too conservative in waiting for messages to arrive the resulting simulator may be prone to deadlock. Finally, a simulator that is correct (it is deadlock free and correctly models the target) can still suffer poor simulation performance if transmission latency between LPs cannot be overlapped with LP execution. Designing high-performance algorithms that satisfy these constraints for CPU hosts has been the central focus of the PDES community since its inception [37].

3.3.1 A Primer on Parallel Discrete-Event Simulation

In PDES, a physical system can be divided into a set of interacting physical processes (PPs). A PDES of this system represents each of these physical processes with logical processes (LPs) that communicate via timestamped messages passed over FIFO communication channels. In effect, any one LP is itself a discrete-event simulator that must process events in timestamp order (the *local causality constraint*). These events can be triggered internally or by messages received from other LPs. The fundamental difficulty in PDES is that an LP may not know that it has not yet received an older message and so cannot safely begin processing another event without potentially violating the local causality constraint. A *synchronization algorithm* is responsible for allowing LPs to make forward progress despite this. There are two broad classes of synchronization algorithm in PDES: *conservative* and *optimistic*.

In conservative synchronization algorithms, LPs wait to process events until they are certain they will not violate the local causality constraint. This conservatism is ripe for deadlock: it simply takes a cycle of LPs with one waiting on the next. Conservative PDES overcomes deadlock by requiring that LPs have non-zero *lookahead*(t_L). That is to say, output messages must only depend on input messages that were received in the past. Armed with this, it becomes possible to detect and resolve deadlock, or avoid it altogether. The Chandy-Misra-Bryant (CMB) [18, 23] algorithm avoids deadlock by introducing *null messages* which indicate the absence of a real message up to the timestamp included in the null message. If an LP with non-zero lookahead is blocked on an input whose latest message has the timestamp t_i, the LP must eventually issue a null message at time $t_i + t_L$ to each of its receivers. This may be sufficient to unblock the receiving LP, resolving the deadlock risk. Failing that, the receiving LP is compelled to send null messages still further in the future and the process repeats. So long as there exists no cycle of LPs with zero lookahead, the CMB algorithm is guaranteed to avoid deadlock: for a complete proof we refer the interested reader to Chandy and Misra [23]. Simulation performance decreases as lookahead approaches zero, as the number of null messages required to avoid deadlock increases [37]. In general, lookahead is derived from underlying properties of the physical process; if sufficient lookahead cannot be captured in the underlying system, conservative PDES algorithms may offer little performance benefit over sequential simulators.

Optimistic synchronization algorithms, first realized by Time Warp [51], permit LPs to speculatively execute events that may violate the local causality constraint. Here, LPs have mechanisms to rollback from mispeculation and to inform downstream LPs of earlier messages that were erroneously sent. In Time Warp, LPs correct for mispeculated messages by sending *anti-messages*, which as the physics analogy would suggest, *annihilate* pair-messages that were erroneously sent in the past. In order to prevent unbounded growth of rollback state and to guarantee forward progress, optimistic simulators have a mechanism to determine *global virtual time* (GVT): the earliest timestamp within the set of all unprocessed events. GVT defines a time across the entire simulator at which the state of all LPs are known non-speculatively [51], rollback state before this timestamp can be reclaimed. While the most basic conservative algorithms can be implemented more simply than optimistic simulators, for many types of physical processes, namely those for which sufficiently long lookaheads cannot be found (either deduced statically or discovered at runtime), optimistic simulators out-perform pessimistic ones. While debate between proponents of conservative and optimistic PDES raged through the 1990s and 2000s, both optimistic and conservative PDES algorithms see contemporary use [38].

3.3.2 Considerations for FPGA-Hosted PDES

FPGAs differ from the multiprocessors and networked hosts studied in the PDES literature in a number of ways:

1. **FPGAs support bespoke, low-latency interconnect between LPs.** Inter-LP interconnect can be tailored for the specific simulation, producing communication channels with very short latencies (a few FPGA cycles). Interconnect is relatively abundant and can support hundreds or thousands of direct links between LPs implemented on the same FPGA.

2. **FPGAs trivially support FIFO communication channels**. Unlike in some conventional PDES hosts, channels between FPGA-hosted LPs are easily made FIFO. LPs do not need to reorder messages, and can assume messages from the same sender arrive in monotonically increasing time-order.

3. **FPGA compute resources are relatively inflexible**. FPGAs are considerably more difficult to program since LPs and other simulation resources must be written in RTL. Existing FPGAs lack native floating-point support which is critical in many PDES application domains. In many cases, it is not clear that FPGA implementation of an LP would be faster than a CPU-hosted one.

The complexity and domain-specific knowledge required to wield FPGAs without any guarantee of improved performance is an enormous deterrent to studying FPGAs as hosts for general-purpose PDES. That said, at least one general-purpose, FPGA-hosted PDES has been built: PDES-A [85]. PDES-A implements an optimistic synchronization algorithm over a regular grid of processing elements (PEs). While it would be possible to implement an RTL simulator using PDES-A, the effective simulation capacity and throughput of such a system would be considerably lower than FPGA-based emulators and prototypes. Where FPGAs hold more promise is in domain-specific PDES that can exploit the features of its interconnect without succumbing to the additional complexity of programming FPGAs.

3.3.3 ASIC Emulation As Domain-Specific PDES

As an application domain, ASIC emulation has a number of properties that make FPGAs well-suited hosts. The first, most obvious reason, is that the programmability challenge is greatly ameliorated by the fact that ASICs are already specified in RTL, and that RTL can be efficiently mapped into an LP on an FPGA. In the case of an RTL block with a single clock, a simple LP implementation clock-gates the block it models and adds state machines to control when to enqueue and dequeue messages, and fire the clock. When an LP clock fires, what would take thousands or millions of instructions in a CPU-hosted simulator is computed concurrently in a single host cycle.

Second, since an ASIC can be decomposed into LPs, each with a relatively small, statically defined number of neighbors, emulation can efficiently leverage FPGA interconnect. High-bandwidth interconnect between LPs removes any need to compress, or otherwise optimize, transmission between them, reducing simulator complexity. Hundreds, even thousands, of cycle-by-cycle traces of messages can be trivially moved between LPs on the same FPGA, a feat that is all but impossible to achieve on a conventional CPU host. LPs can be directly connected to one another with hardware queues, which in the general case, have a single host cycle of "transmission" delay. In some cases, flow-through queues or wire connections may be used to allow combinational transmission of messages. Conversely, while high-bandwidth transmission between LPs can be achieved on a CPU, synchronization (i.e., message transmission time) takes many cycles, which becomes problematic when modeling tightly coupled LPs. The ability to implement low-latency, fine-grained synchronization is perhaps the largest advantage FPGAs offer over conventional CPUs for this application.

Given that FPGAs appear to be good hosts of PDES for ASIC emulation, the question becomes how does one translate an ASIC into a graph of LPs, such that the resulting graph is deadlock free and preserves the behavior of the source RTL while still mapping efficiently onto the FPGA. The natural place to start is with synchronous systems of a single clock domainw, as these are simpler to simulate, yet still useful for doing performance studies on a large class of target machines. These systems can be implicitly timekept—messages are only ever sent at cycle-boundaries—and so synchronization problem associated with general PDES can be greatly simplified. Nonetheless, achieving good performance and deadlock freedom remains a challenge. To solve this, ITDC simulators constrain the behavior of LPs so as to prevent deadlock by construction, by both by exploiting properties of the underlying physical system, and restricting how the target may be legally divided into LPs. We refer to the set of rules that define the behavior of these LP graphs as a *target formalism.*

3.3.4 Target Formalisms for ITDC Simulators

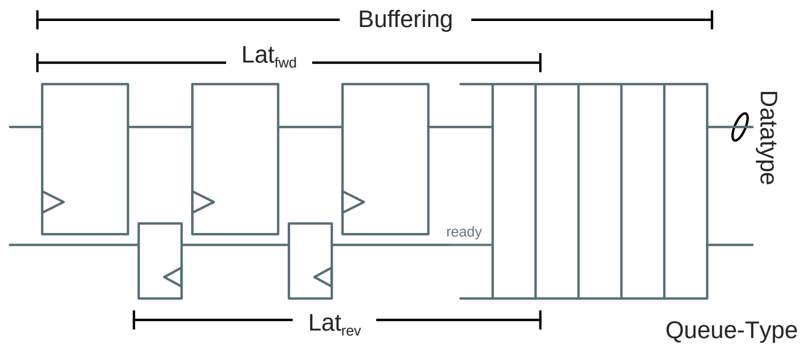
Target formalisms for ITDC simulators are defined by a tradeoff between simulation performance and modeling flexibility. More flexible formalisms lift restrictions on LP definition and permit tighter coupling between LPs whereas coarser-grained formalisms tend to have better simulation performance. We explore three formalisms: the RAMP model (exposed as part of RDL [39]), APortNetworks [80], and LI-BDNs [99]. For the remainder of this dissertation, we will refer to untimestamped messages as *tokens*, the term used more frequently in RAMP-related works, to distinguish them. Having a fixed time-interval between tokens removes the need for conventional conservative PDES deadlock avoidance (i.e, null messages) or detection schemes, instead deadlock avoidance in these formalisms is achieved by-construction by constraining how combinational paths may be modeled between LPs and the conditions under which LP implementations must enqueue and dequeue tokens.

Practically speaking, these formalisms can be employed in two ways. First, they can be used to build *ad-hoc simulators*, where a designer manually writes LP implementations that they stitch together to model a target design that has not yet be built (this includes RAMP-Gold [95], HASim [79], and other RAMP simulators). Here there is no reference system against which the simulator can be validated. Second, they can be used to build hardware emulators, where the LP graph is generated by a compiler that is processing realizable SoC RTL. This RTL in effect a reference system, and so not only can LP implementations be shown to adhere to the rules of the target formalism, but their simulated behavior can be verified.

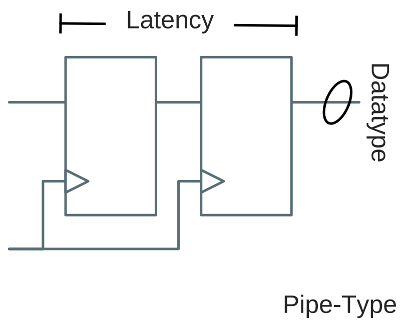
3.3.5 Channel-Bootstrapped Formalisms

The RAMP model [39] and APort Networks [80] are examples of what we call *channel-bootstrapped* formalisms as they rely on the use of special point-to-point LPs called *channels* (in RAMP parlance) to seed the initial tokens of the simulation. Channels are LPs because they model stateful interconnect: it is this statefulness that allows them to send output tokens at time zero before receiving any input tokens. In essence, this is a cycle-level form of lookahead that is sufficient to provide deadlock avoidance.

With that said, RAMP and APortNetworks differ primarily in the granularity of their supported non-channel LPs, or *units* (again, in RAMP parlance). The original RAMP formalism defined units at latency-insensitive boundaries in the target. This would make blocks on the scale of cores or DRAM controllers reasonable candidates for definition as an LP. Channels themselves model latency-insensitive interconnect and generically can be defined by four parameters as shown in Figure 3.1a.



(a) A queue type channel with its four characteristic parameters.



(b) A pipe-type channel. Setting latency to zero models a wire.

Figure 3.1: Two types of channels used in a channel-bootstrapped formalism.

In practice, it is rare to find boundaries in the target where all interfaces are latency insensitive, as often some subset of signals, like reset and interrupts, are driven combinationally or through a set of register stages. APort Networks [80] was designed to support finer-grained LPs that are coupled in this fashion. In their paper, Pellauer et al. use a processor pipeline as a motivating example and divide different pipeline stages into separate LPs. Later versions of the RAMP architecture introduced the same sort of channel, which we call a *pipe* channel, shown in Figure 3.1b. Pipe channels are defined by a single parameter l, corresponding to their latency. Assuming the underlying registers the channel models are not reset explicitly using a target signal, but are instead initialized at time zero to some statically known value, pipe channels provide l initial tokens before they must wait for an input token.

Neither formalism appears to give any treatment to modeling a target-driven reset in channels, as they both rely on time-zero initialization to effectively “reset” target state. We note that if these underlying state elements are synchronously reset, the channels with $l >= 1$ may provide no more than one initial token, as future tokens may causally depend on a reset token. Furthermore, neither of these formalisms was designed to support asynchronous reset as all events are synchronous to an implicit clock. Asynchronous reset mandates that that even the first token emitted by a pipe-channel with $l > 0$ would depend causally on reset at time zero. A pipe-type channel with $l = 0$ is a degenerate LP that models a wire between the producer and consumer LP, coupling them combinationally. Wire-type channels model no target state and merely relays tokens from producer to consumer.

In both formalisms, a unit simulates a single target cycle of execution by dequeuing an input token from each of its input ports, and enqueuing a new output token into each of its output ports. The simplest unit implementation, and the one prescribed by APort Networks, waits for all input tokens to arrive and all of its output ports to be ready to accept tokens before it may fire. Then the unit computes its state updates and simultaneously enqueues a single output token into each output port, and dequeues a token from each input port. Figure 3.2 demonstrates the execution of a two target cycles in a channel-bootstrapped formalism.

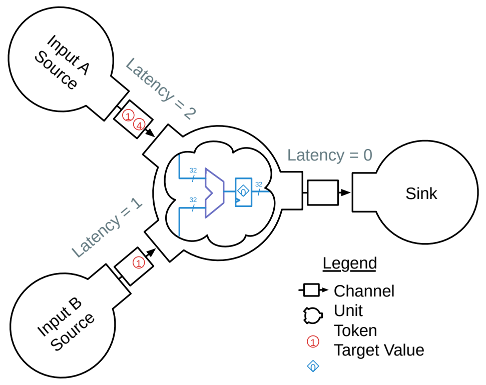
If the simulation designer uses exclusively non-wire-type channels, both formalisms are free of deadlock as there exists no unit whose output at cycle t depends on the output of another unit also at cycle t. From a conservative PDES perspective, every cycle of LPs has a non-zero lookahead, and given that all LPs send tokens at every timestamp (obviating the need for null messages) deadlock is avoided by construction. In these simulators, all units can execute cycle t concurrently which, assuming no other sources of delay, permits the simulator to run at unity FMR. As is the case with latency-insensitive boundaries, it is not always possible to define LPs at registered boundaries. The use of wire-type channels (i.e. combinational coupling between LPs with $T_L = 0$) introduces two principal challenges: at best it increases simulator FMR, and at worst it may introduce time-zero simulation deadlock. To illustrate each of these effects, in Figure 3.3 we consider a latency-insensitive interface between two units implemented three different ways: with a queue-type channel, with one pipe and one wire-type channel, and with two wire-type channels.

In the absence of deadlock, wire-type channels tend to increase FMR as in general a combinational path that spans N, units will take N cycles to execute. Note that other channels connecting these units tends to prevent pipelining multiple cycles of target execution (i.e., this latency cannot be overlapped with the simulation of younger target cycles).

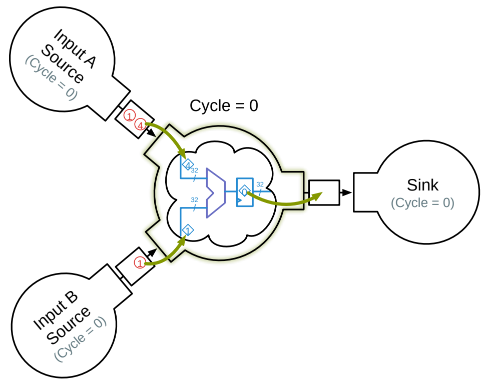
Deadlock is the more pressing challenge. Wire-type channels remove the aforementioned property that all cycles of LPs have non-zero lookahead. Here there are two situations that may emerge. If the target itself has a combinational loop, the behavior of the physical system is itself undefined, and simulation deadlock is unavoidable. We note it is possible to have “false” combinational loops but many tools reject these patterns, and so it is not unreasonable to ban them in these simulators as well. That said, as a matter of practice, hardware designers know to avoid combinational loops. The second challenge arises from poor LP implementation, such as the implementation prescribed by APort Networks, that attempts to enqueue and dequeue multiple tokens simultaneously. Mandating this implementation has the effect of introducing a dependency between all output tokens on all input tokens, even though there they may be no combinational path between them in the underlying target. Given its definition of a unit, APort Networks do not give the correct set of constraints to avoid deadlock as a graph that contains a cycle of wire-type channels—that do *not* represent a combinational loop in the target—deadlocks at time zero.

3.3.6 Latency-Insensitive Bounded-Dataflow Networks (LI-BDN)

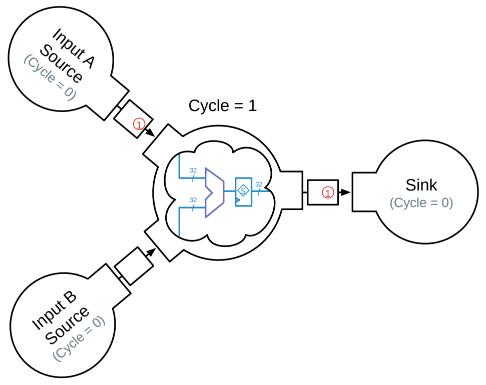
Unlike channel bootstrapped formalisms, LI-BDNs avoid deadlock by imposing different constraints on the implementation of LPs based on the underlying hardware they model, and remove the need for specialized channels to bootstrap simulation. Since connections between LPs always represent wire-type connectivity, LI-BDNs impose no constraints on how a synchronous target is divided into LPs. LI-BDNs were originally developed as a more



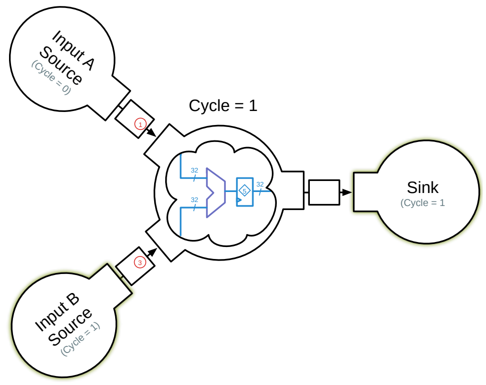
(a) Initial state of the graph.



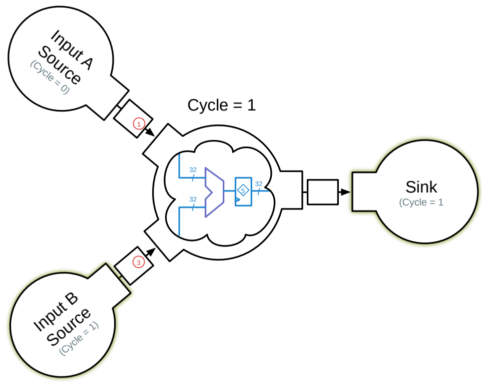
(b) With all input tokens available, the unit advances to cycle one.



(c) The state after one firing. The unit stalls waiting for a B-input.

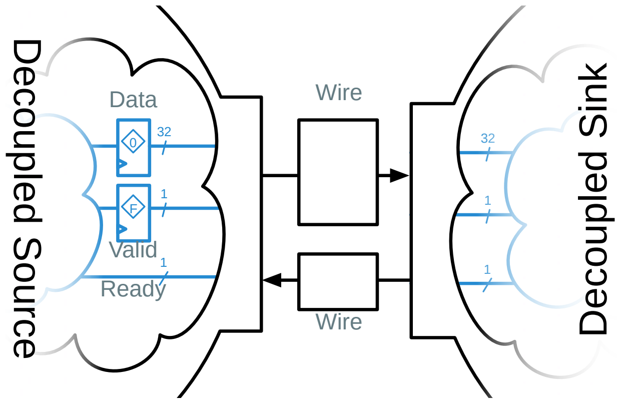


(d) Input B source fires, producing the needed token.

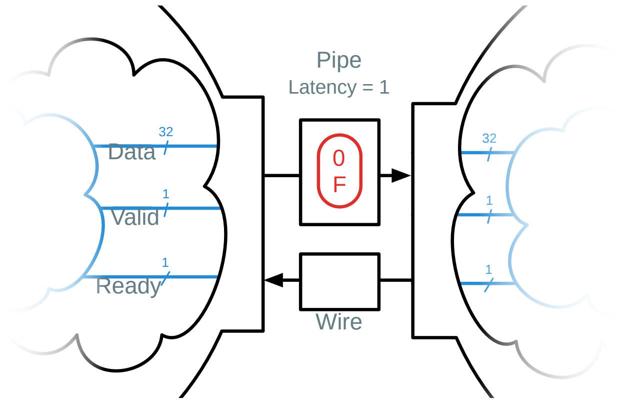


(e) The state after two firings.

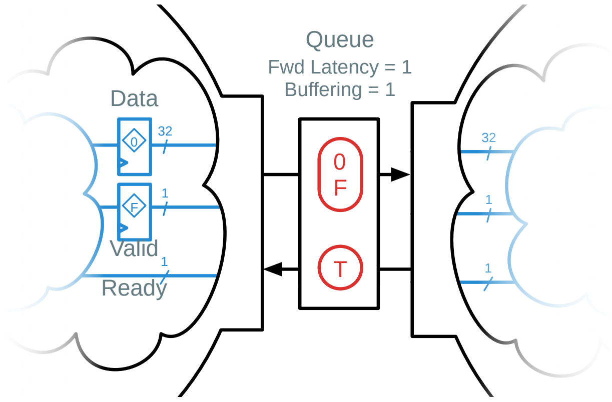
Figure 3.2: A 32-bit adder model and environment simulating two cycles of target time.



(a) Two-wire type channels. This deadlocks an APort Network, as neither producer nor consumer unit can fire before the other.



(b) One pipe-type channel (carrying the payload) and one-wire type channel (carrying backpressure). This leads to $FMR >= 2$, with the decoupled-source unit firing first to produce a backpressure token that is subsequently consumed by the decoupled-sink unit.



(c) A queue-type channel. This allows both models to fire simultaneously, but may only be used if it is acceptable to model a target queue in the channel.

Figure 3.3: Three potential channel-modeling decisions for a latency-insensitive target interface between two units.

flexible abstraction for building latency-sensitive designs that preserve the RTL timing of a synchronous circuit, extending the work of Carloni et al. in the Theory of Latency Insensitive Design [21], however the authors identify that this problem is analogous to constructing an ITDC simulator from synchronous ASIC RTL. From a PDES perspective, we will see that LI-BDNs avoid deadlock by ensuring that units (LPs) themselves always exploit single-cycle lookahead in their reference physical processes when it is available.

In the terms of Vijayaraghavan et al.[99], what we have loosely referred to as a synchronous block of RTL thus far is defined as a synchronous sequential machine (SSM). Vijayaraghavan et al.[99] formally define an SSM:

> An SSM is a network of combinational operators or gates such as AND, OR, NOT, and state elements such as registers, provided the network does not contain any cycles which has only combinational elements.

Missing from this definition is any treatment of multiple clock domains, different types of state elements, asynchronous set or reset. Inherent to the name "synchronous", this formalism forbids their existence – in an SSM *all* state updates occur simultaneously. While this limits the applicability of their approach, much of a modern SoC, such as the islands of a GALS-based design, locally meet this definition. With its simple state-update semantics, any SSM can be easily converted into a *patient SSM*: the equivalent circuit whose state update can be controlled with the assertion of a global enable signal. One way this conversion can be achieved, proposed by the authors, is by disabling register updates and masking off RAM write-enables when this control signal is unset. Alternatively, we note that the SSM may be clock-gated.

With an SSM defined, we can now define an LI-BDN, and how they can be used to implement any SSM in an latency-insensitive manner. An LI-BDN is a dataflow network of latency-insensitive nodes whose edges represent FIFO communication channels with non-zero, but finite capacity. The latency-insensitive nodes of the graph are referred to as *Primitive LI-BDNs* (in PDES terms, these are LPs), which is to say, they are not themselves subgraphs of more than one node. A LI-BDN is said to *implement* an SSM, if it *partially implements* the SSM, and is deadlock free. In the terms of a hardware engineer, partial implementation (PI) implies that the token trace of outputs from the ports of the LI-BDN "matches" the per-cycle trace of outputs from the SSM. Vijayaraghavan et al.[99] formally define PI:

> A BDN R partially implements an SSM S iff
>
> 1. There is a bijective mapping between the inputs of S and [the input tokens of] R, and a bijective mapping between the outputs of S and [the output tokens of] R.

2. The output histories of S and R match whenever the input histories match, i.e.,

$$\begin{aligned} &\forall n > 0 \\ &\quad I(k) \text{ for } S \text{ and } R \text{ matches } (1 \leq k \leq n) \\ &\Rightarrow O(j) \text{ for } S \text{ and } R \text{ matches } (1 \leq j \leq n) \end{aligned}$$

All SSMs have many possible partial implementations, though some implementations deadlock when composed into larger graphs. Consider the unit implementation prescribed by APort Networks: it produces valid partial implementations of source RTL, however it cannot implement a larger SSM when it is composed with other similar implementations without the use of non-wire-type channels (which are in essence LPs that remove zero-cycle loops between units). Conversely, LI-BDNs guarantee deadlock freedom by adhering to two additional properties. The first is the *no extraneous dependencies* (NED), property which defines when an LP is obligated to enqueue output tokens. Vijayaraghavan et al.[99] formally define NED:

> A primitive BDN has the NED property if all output FIFOs have been enqueued at least $n-1$ times, and for each output O_i, all the FIFOs for the inputs in *CombinationallyConnected*(O_i) are enqueued n times, and all other input FIFOs are enqueued at least $n-1$ times, then O_i FIFO must eventually be enqueued n times.

One intuitive explanation of NED is that it exploits the same observation that permits a pipe-type channel to send output tokens before receiving any input tokens: any output driven only by state in the LP (i.e., outputs not combinationally coupled to an input) can enqueue an output token for cycle t before receiving any input tokens for cycle t. NED broadens this to apply to logic cones fed not just by state elements but those that depend only on a subset of the LP's inputs.

The second property an LI-BDN must satisfy is the *self-cleaning* (SC) property. SC defines when an LP is obligated to dequeue input tokens. Vijayaraghavan et al.[99] formally define SC:

> A primitive BDN has the SC property, if when all the outputs are enqueued n times, all the input FIFOs must [eventually][4] be dequeued n times, assuming an infinite source for each input.

While the nodes of an APort Network do not always satisfy NED, they do satisfy SC. At its heart, SC provides an assurance that LPs drain their input FIFOs, allowing those FIFOs to have a bounded size. We note that if an LP fails to satisfy SC, it will likely fail to partially implement its SSM unless its outputs are completely independent of the inputs

[4]Clarified in [98].

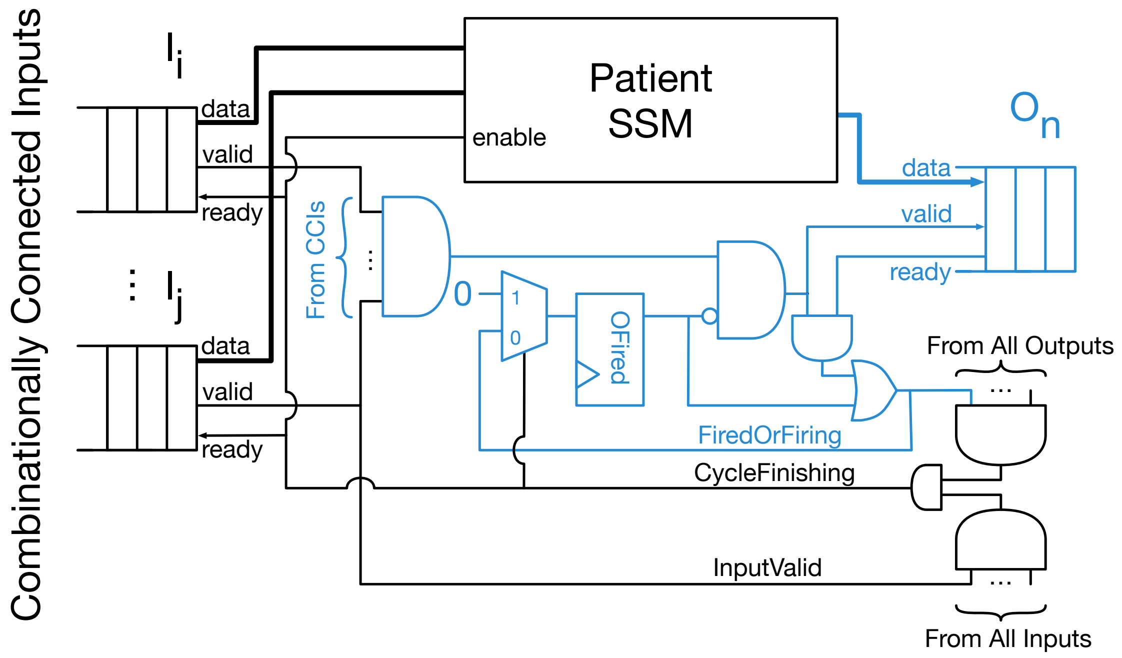


Figure 3.4: A wrapper-module-based conversion of a patient SSM into a primitive LI-BDN. The logic in blue is generated per-output, whereas logic in black is instantiated once.

from which it is failing to dequeue. In this case having unbounded FIFOs would suffice to prevent deadlock.

Of particular concern to this dissertation is a method through which an SSM can be converted into a primitive LI-BDN. Vijayaraghavan et al. describe a means that uses a wrapper module around a patient SSM. We show a modified version of this wrapper circuit, for a single output, in Figure 3.4 and contrast it against an equivalent APort Network wrapper in Figure 3.5. Note in the LI-BDN wrapper, outputs are permitted to fire before all input channels hold valid tokens for the current cycle. We name the SSM-enable signal `CycleFinishing` to reflect the fact that, unlike in the APort Network wrapper, a target cycle can be executed over multiple host cycles as combinational paths resolve and new input tokens arrive. `OFired` registers ensure exactly one output token is enqueued per target cycle. `CycleFinishing` is fed by the `OFired` registers in addition to the same signals that feed `TargetFire` in the APort Network wrapper and so will have greater logic delay. The wrapper circuit suggested by Vijayaraghavan et al. [99] drives `CycleFinishing` with the AND-reduction of all `OFired` registers – this reduces logic delay at expense of increasing FMR by 1. Conversely, both wrapper circuits in Figure 3.4 and Figure 3.5 can run at unity FMR. Both circuits are highly amenable to automatic generation with a FAME compiler. In addition to the ability to translate an SSM into a patient SSM, the LI-BDN wrapper requires only a mechanism to analyze combinational dependencies in the underlying SSM.

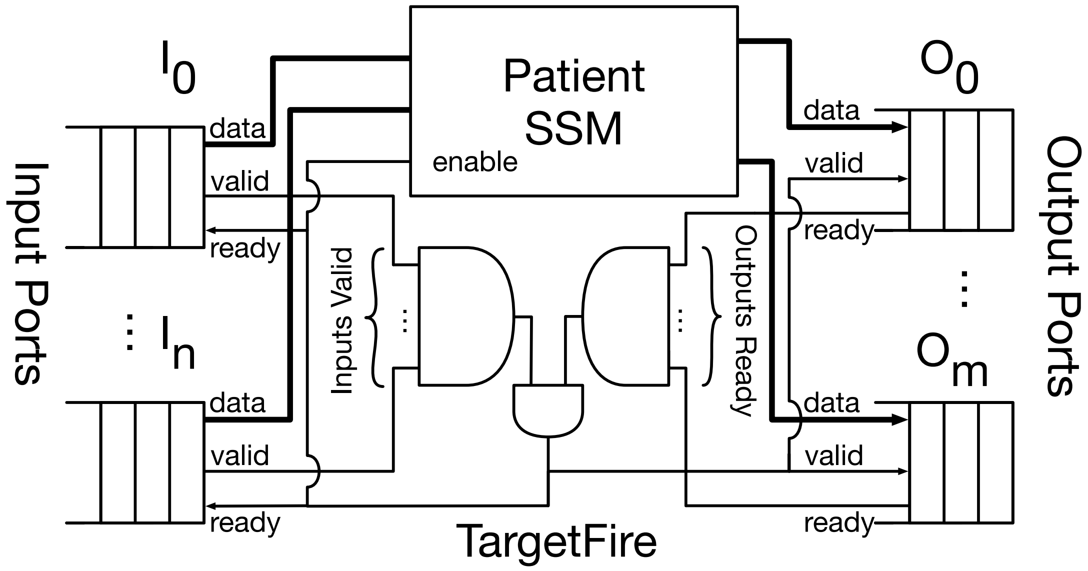


Figure 3.5: A wrapper-module-based conversion of a patient SSM into an APort Network unit.

3.4 The MIDAS Compiler & Generated Simulator Microarchitecture

MIDAS, which we introduced in Section 2.3.1, was our first attempt at building a FAME compiler. Here we describe in greater detail how it was implemented. Specifically, we direct the reader to the version of MIDAS released as part of FireSim 1.5.0, which differs modestly from earlier iterations described at CARRV2017 [56] and Donggyu Kim's dissertation [55] and is the last version of the compiler before the introduction of Golden Gate features.

MIDAS generates ITDC simulators directly from ASIC RTL. These simulators implement a channel-bootstrapped formalism with all target graphs having a star topology. We show an example graph for a Rocket-Chip-based system, the most commonly used target generator in MIDAS, in Figure 3.6. The *hub unit* is transformed from ASIC RTL that has been elaborated by a user-provided Chisel generator. This RTL does not represent a closed system: it exposes I/O interfaces that must be tied to units that model those devices. These units, which form satellite nodes in the graph, we call *endpoint* units. MIDAS-generated simulators are designed to run on hybrid CPU-FPGA hosts with the expectation being that certain tightly coupled endpoints, such a DRAM timing models, will be written in RTL and hosted on the FPGA, while other higher latency, less performance-critical ones, such as UART and block device models, can be written in software hosted on the CPU. We refer to the CPU-hosted component of the simulator as the *driver*. To avoid deadlock and provide good simulation FMR, MIDAS injects queue-type channels (configured to model a two-deep, fully decoupled FIFO) on all decoupled interfaces, and one-cycle latency pipe-type channels on all others.

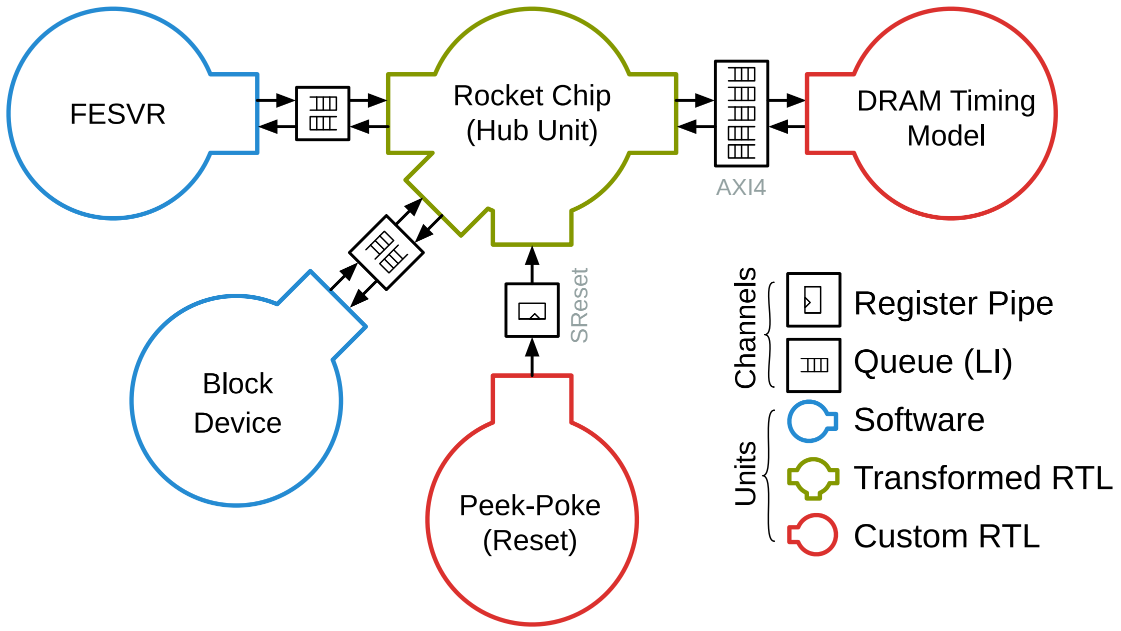


Figure 3.6: The graph of a typical Rocket-Chip-derived system.

Assuming units do not stall internally, MIDAS-generated simulators can run at unity FMR. We show an example simulator mapped to an EC2 F1 host in Figure 3.7.

3.4.1 Endpoint Units

Topological position aside, endpoint units differ from the hub in that they are not transformed from ASIC RTL and instead are written by hand, and thus, are not exact models of the desired target system. Endpoints consist of an *endpoint module*, which resides on the FPGA and drives the channels bound to the hub unit, and an *endpoint driver*, which is linked into the main simulation driver and hosted on a CPU. Even the purest “CPU-hosted” units require an endpoint driver that implements transport for token streams moving on and off the FPGA. The one exception to this rule occurs when FireSim stitches together larger targets by adding software units to represent parts of the network. These are hosted purely on a CPU on EC2 instances that may have no FPGA attached. Similarly, FPGA-hosted models tend to have a driver that is responsible for configuring the endpoint module before simulation commences and for periodically polling instrumentation registers.

Endpoints can leverage three types of resources:

1. **CPU-mastered MMIO.** Here, modules expose 32-bit registers that can be written to and read from the driver. This is typically used to expose configuration parameters that the driver initializes before simulation commences, and to expose instrumentation that can be polled during simulation. Generally, MMIO cannot provide enough bandwidth to support transporting token streams, but for low-bandwidth, loosely coupled

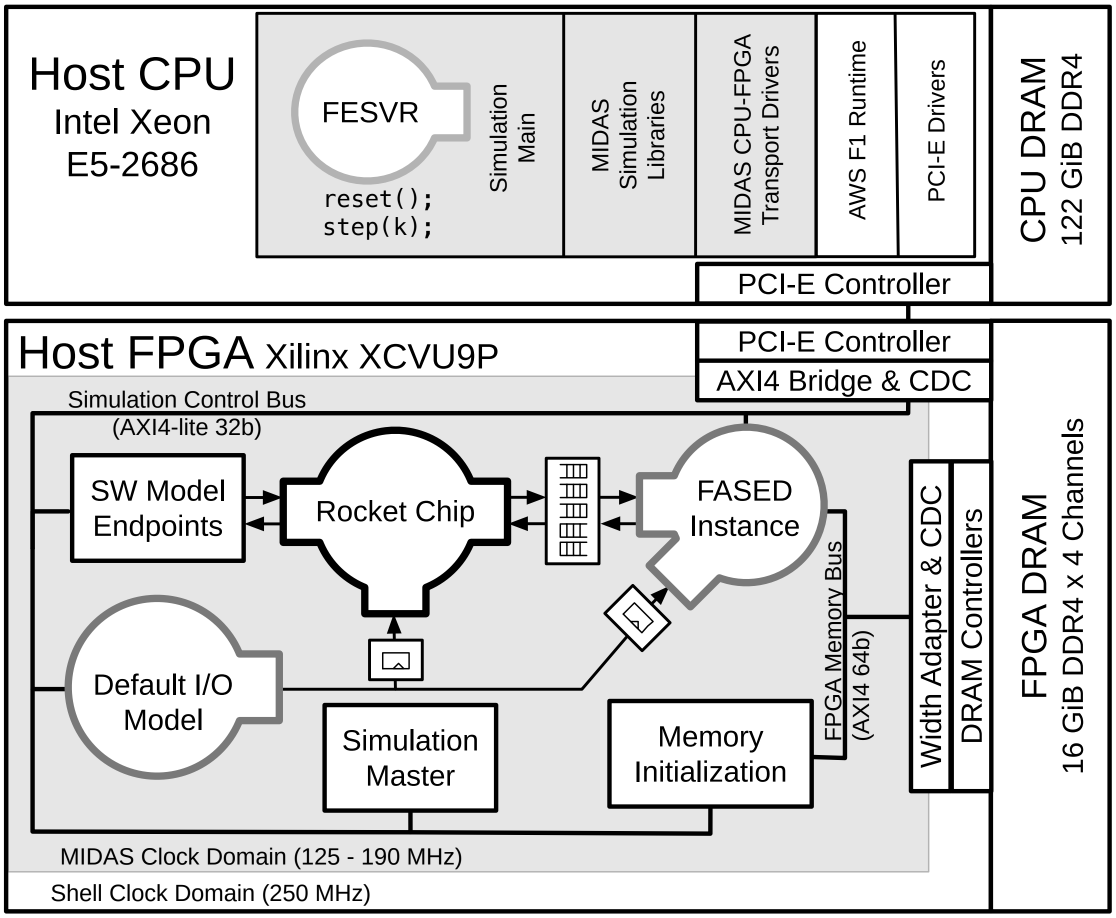


Figure 3.7: An example simulator mapped to an Amazon EC2 F1 host (`2x.large`).

endpoints it can also be used to pass coarser-grained transactions. The FireSim block device endpoint uses this technique.

2. **CPU-mastered DMA.** To support moving complete token streams, endpoints can bind to CPU-mastered DMA. Here endpoint modules expose hardware FIFOs to which the driver can make bulk (up to 4 KiB) reads and writes. If the endpoint is not closely coupled to the hub, token transport can be overlapped with simulator execution: the NIC endpoint relies on this to provide good FMR in networked simulations [53]. CPU-mastered DMA is also used to drain token streams from instrumentation endpoints. Since these endpoints do not drive tokens back to the hub, it is only necessary to provide sufficient DMA read-bandwidth to achieve good FMR. Examples of endpoints that use CPU-mastered DMA include the synthesized `printf` endpoint and the TracerV RISC-V instruction trace collection endpoint.

3. **FPGA DRAM**. Some units require more local storage than FPGA fabric can provide, but cannot afford the round trip latency of using CPU memory or disk. Instead, MIDAS

provides a simple means to let endpoints drive the FPGA DRAM memory system. MIDAS's memory timing models, both those described in the CARRV publication [56] and later by FASED [16], use FPGA DRAM as a backing store for a timing model that is implemented in FPGA fabric.

Endpoint drivers interact with their associated module using four methods exposed by the main simulation driver: `simif_t::read` and `simif_t::write` drive CPU-mastered MMIO; `simif_t::pull` and `simif_t::push` are used to drive CPU-mastered DMA.

3.4.2 Simulation Driver

The simulation driver is written in C++ and instantiates an endpoint driver class for each endpoint in the system. Once the FPGA has been flashed, the driver can be launched. Simulation proceeds then proceeds in three phases:

1. **Initialization.** This provides a window for endpoint drivers to do DMA and MMIO before the simulation commences. Here the driver optionally zeros-out or initializes FPGA DRAM, before calling each endpoint driver's `init` method, which sets up configuration registers on the FPGA that may control timing parameters or instrumentation modes.

2. **Execution.** Here the driver invokes each endpoint's `tick` method in a tight loop, in round-robin order. Drivers issue MMIO and DMA requests attempting to make forward progress without blocking.

3. **Teardown.** Simulation ends once any endpoint driver calls for termination. At this point, the driver calls each endpoint's `finish` method, which gives the endpoint a final opportunity to do FPGA DMA and MMIO. This typically includes reading final FPGA-hosted instrumentation values and committing simulation output to disk.

3.4.3 Time Control

Immediately after FPGA programming and reset, channels are ready to produce initial tokens and thus units are free to begin executing. Generally, endpoints that require configuration wait for an MMIO register to be set during the driver's initialization phase before acting on token streams. In the absence of user-provided endpoints, MIDAS prevents the hub model from free-running by mandating that all target designs have a synchronous reset that is driven by a special *peek-poke* endpoint. The peek-poke endpoint ties off all unconnected I/O on the target to memory-mapped registers that can be driven to specific constants during initialization, or periodically changed during simulation. The driver can then "step" the hub by controlling the emission of reset tokens by the peek-poke endpoint. Unlike for other endpoints, channels between the peek-poke endpoint and the hub are wire-type by default: enqueuing k reset tokens permits the hub to execute no more than k cycles.

During simulation teardown, the simulator estimates the number of target cycles executed by reporting the number of reset tokens enqueued by the peek-poke endpoint. This relies on the observation that the hub must consume a single reset token for each target cycle of execution. This is not exact since the hub may not have drained all tokens in the reset channel (it is in the past relative to the peek-poke endpoint) and, additionally, other endpoints connected to the hub may be relatively more or less advanced in simulation time.

3.4.4 MIDAS's Generator Side Integration

MIDAS functions by providing an alternate generation flow for a Chisel generator. In their Scala application, the Chisel user lazily instantiates their module class and calls out to Chisel elaboration as they would normally. This produces a FIRRTL circuit and annotations for what will become the hub model. Typically the lifetime of a Chisel module class ends after elaboration, however the lazy reference to the class allows the user to extract the names and Chisel types of the I/O interfaces of the top-level module after elaboration. This is critical as MIDAS determines how to bind endpoints to the hub unit by querying a user-provided map of Chisel-type to endpoint-module for each of the extracted I/O interfaces. Finally, the user invokes MIDAS. To the compiler they pass the FIRRTL circuit and annotations, the extracted I/O list, separate lists of FIRRTL transforms to run on the elaborated RTL and then the complete simulator, and a parameters object which provides the aforementioned map and enables various compiler features, such as `printf` and assertion synthesis.

3.4.5 The MIDAS Compiler Flow

We show the MIDAS compiler flow in Figure 3.8. Before running core transformations, MIDAS schedules user-provided target transforms. Here, FireSim adds dedicated passes to remove unsupported Verilog black-boxes and replace asynchronously reset registers with synchronously reset ones.

Next MIDAS runs instrumentation transformations, including `printf` synthesis and assertion synthesis (these were initially presented in DESSERT [57]), if they have been requested by the user. These transformations "synthesize" `printf` and `stop` FIRRTL IR nodes by replacing them with connections to new top-level outputs. `stop` statements are replaced with a boolean that is asserted on cycles the stop condition is asserted. Similarly, `printf` statements are replaced with an aggregate that contains the arguments for the `printf`'s format string and an enable signal indicating that the `printf` would trigger on that cycle. Like the other I/O interfaces, these new outputs will be bound to a single assertion and `printf` endpoint later in the compiler.

Now MIDAS runs its FAME1 transform, transforming the RTL (an SSM) into a unit. To this end, MIDAS generates the APort Network wrapper module shown previously in Figure 3.5. MIDAS converts the SSM into a patient SSM by injecting clock-enables on all registers (this introduces a 2:1 multiplexer on register inputs) and by masking FIRRTL memory write enables. The transform divides and maps the SSM's I/Os into token ports by

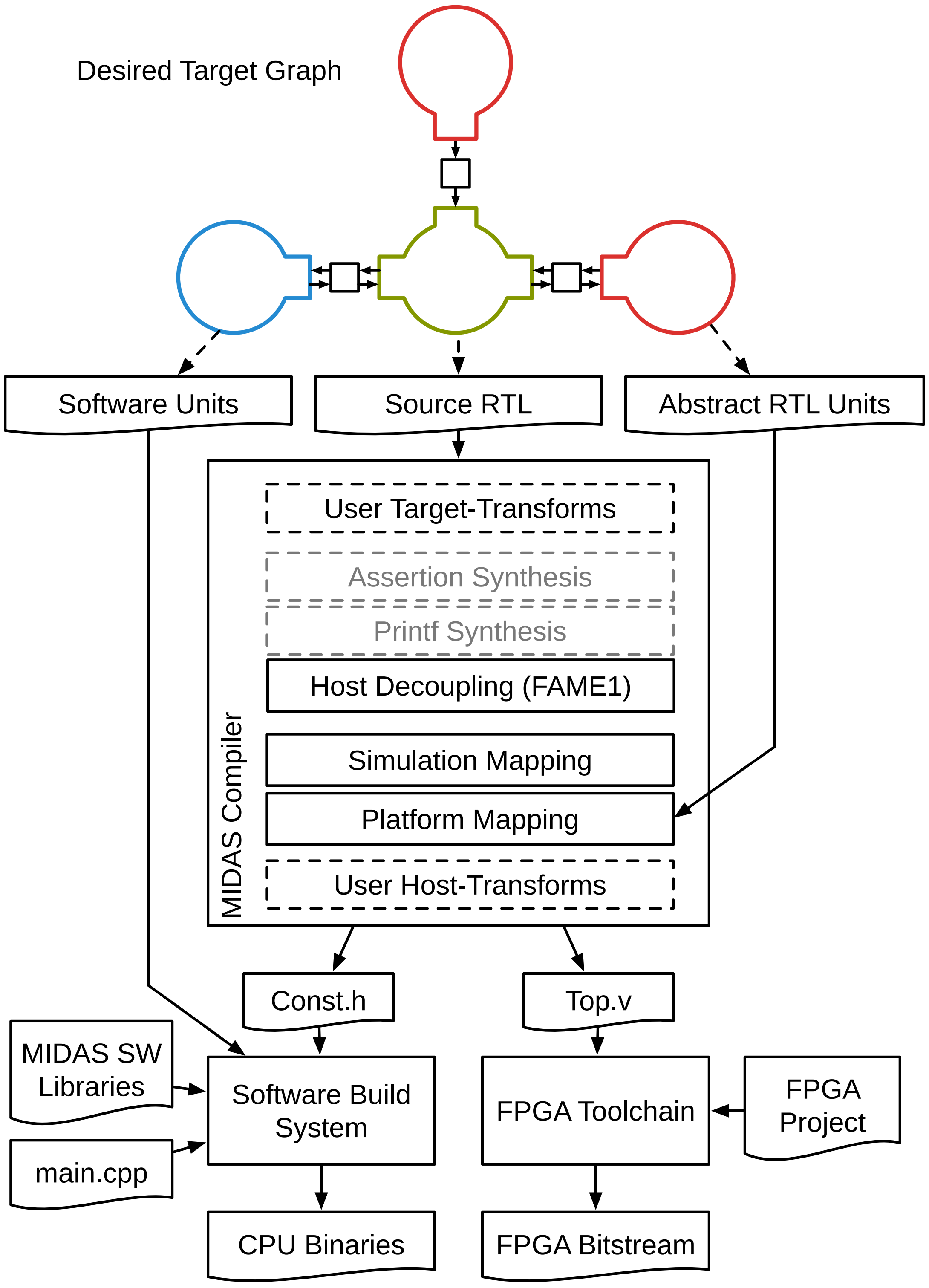


Figure 3.8: The end-to-end flow of a MIDAS compilation.

recursing through their corresponding Chisel types. Decoupled interfaces (latency-insensitive interfaces that use a ready-valid handshake) are split into a forward and reverse port, all other interface types are given a separate port per leaf-element.

To complete the simulator, MIDAS then wraps the hub unit with two layers of Chisel-generated modules. In *Simulation Mapping*, MIDAS again recursively walks the Chisel types of the I/O, this time to generate the channel implementations. Wire and pipe-type channels are implemented using a queue whose payload is the underlying hardware type of the target interface. The endpoint and hub can decouple as many cycles as these queues are deep. On host reset, pipe-type-channels pre-enqueue a single token (with all bits cleared) to bootstrap the simulation. Queue-type channels use a more sophisticated implementation that drives forward and reverse token streams. The queue channel implementation allows endpoint and hub to decouple under more specific runtime conditions (for example, while a queue is not full it can continue to accept forward tokens and produce reverse tokens on its enqueue interface).

In *Host Platform Mapping*, MIDAS recursively walks the Chisel-types of the I/O a final time and attempts to find a provided endpoint for each type using the Endpoint Map. If the map is defined on that type, MIDAS instantiates the endpoint module, and connects its token-passing port (`HostPort`) to the freshly generated channels. All remaining unbound interfaces are then bound to the peek-poke endpoint. Once all endpoints have been elaborated, MIDAS binds their MMIO interfaces and DMA interfaces (if present) to the simulation control and DMA buses, respectively, using two AXI4 arbiters. Memory-timing models have their host-memory interface driven to DRAM memory interfaces, this time through an $M + 1 : N$ AXI4 crossbar, where M is the number of target memory models and N is the number of available host memory channels. The extra master port is driven by a memory initialization module (`loadmem`), which bridges the simulation control bus and memory bus and gives the driver the means to read and write to FPGA DRAM. The parameters of the host platform, notably the width of these buses and the number of memory channels are defined in the user-provided parameters object. At this point MIDAS generates a C++ header with all of the driver-required metadata for the simulator. Each endpoint module emits a fragment which contains its allocated MMIO and DMA addresses, as well as additional constructor parameters required by its endpoint driver.

In a final step before Verilog emission, MIDAS schedules the user's *host transforms.* This provides a means to programmatically modify simulator RTL to further customize it for a host platform. In FireSim, the `AutoILA` transform is run here to plumb out annotated signals to a Xilinx Integrated Logic Analyzer (ILA), which is required for EC2 F1 hosts due to restrictions in Amazon's compilation flow. Finally, the FIRRTL optimization passes are run and the simulator Verilog is emitted.

At this point, the user links the generated header into the simulation driver, and compiles the generated Verilog into a FPGA shell project. FireSim's build system automates much of this, and provides an FPGA shell project for EC2 F1 FPGAs. The FireSim manager's `buildafi` will batch out bitstream builds to remote hosts on EC2. Similarly, its `infrasetup` process compiles the simulation driver and runs a number of other required tasks to prepare for simulation, such as flashing the host FPGA.

3.5 Reviewing MIDAS's Limitations

At the end of the previous chapter, we identified two limitations with MIDAS that we aimed to address in Golden Gate. The first was that it did little to optimize FPGA resource utilization, precluding its use for systems of non-trivial scale. We made our first attempt at addressing this in Golden Gate in its eponymous ICCAD 2019 publication [70]. We did this by switching to an LI-BDN formalism, building in compiler support to identify and extract target submodules, and supplying hooks to later replace those submodules with optimized primitive LI-BDN implementations. We expand on the design of this initial version of Golden Gate in the next chapter (Chapter 4). Continued work on FAME-compiler-driven optimizations is the subject of Albert Magyar's dissertation [71].

Tackling the second challenge of supporting targets with realistic clocking and reset schemes required yet another rethinking, as the SSM assumption fundamental to LI-BDN and APort Network formalisms no longer applies. In Chapter 5, we describe a simple extension to Golden Gate to support multiple target clock domains under the assumption that they have fixed, rational relationships to one another. Here, we can still treat the target as synchronous but to a fast, virtual clock whose frequency is the least-common multiple of actual target clocks. This assumption permits simulators to remain implicitly timestamped.

To build a true emulation environment capable of simulating the broad class of asynchronous events that manifest in realistic SoCs, including asynchronous reset, clock generation and switching, and certain nonidealities like clock drift and jitter, we lifted the rational clock assumption and introduced explicit timestamping into a sub-graph of the simulator. Our prototype, described in Chapters 6 - 8, uses a hardware implementation of the Chandy-Misra-Bryant algorithm to avoid deadlock. This explicitly timekept portion of the graph coexists with SSM optimizations which can be applied locally to parts of the design where the SSM assumption holds.

Chapter 4

Golden Gate: An Optimizing FAME Compiler

To the best of our knowledge, Golden Gate (née MIDAS II) is the first open-source, optimizing FAME compiler. The content in this chapter loosely mirrors that of our 2019 ICCAD publication [70], but better contextualizes the design of the compiler against MIDAS; describes features, like target-to-host bridges, not discussed in the ICCAD publication; and elaborates on implementation details pertinent to the later chapters of this dissertation. For a second perspective, with expanded discussion on the optimizations themselves and how they were verified, we direct the interested reader to Albert Magyar's dissertation [71]. Code references in this chapter will refer to sources in FireSim version 1.8.0[1].

4.1 Design Objectives

We had the following objectives in mind when we set to redesign MIDAS:

1. **Maintain FireSim feature completeness.** We wanted the ability to drop in Golden Gate as a replacement for MIDAS in a future FireSim release, with the only user-facing difference being the availability of new optimizations. As a side effect of a more sophisticated compiler we expected compile times would increase, however we wanted unoptimized simulators to have approximately the same performance and resource utilization as those generated by MIDAS.

2. **Minimize user modification of ASIC RTL.** It is all too easy to provide a crutch to the compiler by requiring that the user make RTL changes to expose optimization opportunities. The difficulty is these changes can be invasive: they may introduce new bugs, adversely affect ASIC QoR, or worsen source maintainability. While we expect there may be times where these types of changes are desired, we made no such changes

[1]https://github.com/firesim/firesim/tree/1.8.0

in the Rocket Chip or BOOM codebases to implement the optimizations described in our ICCAD paper.

3. **Provide a mechanism to rigorously verify the optimizations.** RAMP-style, multi-cycle resource optimizations introduce considerable complexity into the simulator, making the resulting system harder to debug effectively. A buggy optimization can introduce simulation deadlock or false bugs in the target design. More insidiously, an optimization may produce a functionally correct simulator with slightly different target-timing behavior (e.g., a latency-insensitive transaction may be delayed an extra cycle), introducing difficult-to-detect performance discrepancies. We wanted users to trust the compiler wasn't changing their design beneath their feet.

4. **Enable the use of FireSim as a library for hardware emulation.** MIDAS and FireSim had developed in isolation of the chip-design projects ongoing at UCB-BAR, and used custom forks of many of the same hardware and target-software libraries. This made it challenging for chip designers to use FireSim without help from a developer to make their design "FireSim-compatible". We wanted to make it possible to include FireSim in a chip-like project—specifically, in what would would become Chipyard [3]—to support seamlessly pushing real designs through an emulation flow.

Our desire to maintain FireSim feature completeness made it clear from the outset that a complete redesign of the compiler was untenable in a reasonable timeframe. Instead, Golden Gate was developed as a series of reimplementations of key phases of MIDAS. This let us continually build functional emulators of the same designs supported in mainline FireSim. We started with introducing support for optimizations while maintaining the existing endpoint support and compiler interface (described in Sections 3.4.1 and 3.4.4 respectively).

To enable optimizations, we considered a few possibilities such as leaning on the endpoint system and user modifications to the target RTL, which would imply abandoning our second objective, or using specialized transformations to implement optimizations as modifications to the hub model, which would make it more difficult to achieve our third objective. Ultimately, what we settled on was an LI-BDN-based compiler organization that leverages extensive target module-hierarchy modifications to wrap and isolate optimization candidates in their own modules. These modules become the reference SSMs for each unit. They can be FAME-transformed into a primitive LI-BDN and reimplemented without consideration for how they are connected to the rest of the simulator. Then, the unit and its corresponding reference module can be fed into a verification flow to show the resulting unit partially implements the reference SSM. In principle, this could be applied to every unit-SSM pair in the network to show the simulator is a valid LI-BDN implementation of the source design.

The flexibility LI-BDN provides regarding how the the target is divided into units was a critical factor in our choice to adopt it over a channel-bootstrapped formalism. LI-BDN removes the explicit need for channels and permits modules (and thus resulting units) to be combinationally coupled which substantially widens the scope of potential optimizations.

While we acknowledge that forcing optimizations to apply only at registered boundaries between units (a requirement of a channel-bootstrapped formalism) would produce simulators that could sustain greater FMRs, this would add complexity to the compiler and could make many optimizations infeasible. We also note that these types of "channel-like" optimizations are not fundamentally precluded by the LI-BDN formalism. For example, when an edge between two units is driven by an output port with no combinational dependency on an input, it may be possible to implement that edge using a flow-through queue to save a cycle of transmission latency.

Moving to an LI-BDN formalism resolves a number of other modeling challenges MIDAS users face. Requiring that non-wire-channels be injected between endpoints and the hub unit is too restrictive in some cases where the user wishes to model a combinational path that propagates through an endpoint. Another challenge was defining the reset semantics of these stateful channels. MIDAS relies on FPGA programming to properly initialize these channels, however they are not held in target reset during simulation: it is purely coincidental that the hub unit does not spuriously enqueue target data into queue-type channels while parts of the hub are under target reset. For a time, MIDAS would broadcast target reset tokens to channels and endpoints, but this too is fraught, as reset signals driving registers in the hub may be different from this special-cased global reset. Instead of requiring that user specify this reset for each channel, LI-BDN can sidestep these issues entirely by removing the need for stateful channels and allowing the compiler to assume fewer things about the target.

The second phase of Golden Gate's development addressed the fourth objective, and required a redesign of the compiler interface and endpoint system. Here, we decoupled Golden Gate from target elaboration. Instead, Golden Gate accepts a FIRRTL file and annotations. *Target-to-Host Bridges*, or bridges for short, replace endpoints and are instantiated throughout the target module hierarchy during target elaboration. Unlike MIDAS, Golden Gate does not match on the Chisel I/O of the top-level module. Instead, it finds annotations (`BridgeAnnotations`) bound to the aforementioned modules that provide the class name of a Chisel generator Golden Gate should invoke to replace the black box.

4.2 Compiler Input & Target Specification

Golden Gate accepts a *closed* description of the target consisting of FIRRTL and annotations. We show a pictoral example of this using a typical Chipyard-generated SoC in Figure 4.1. By "closed" we mean that the top-level module has no I/O: the user instantiates their chip in a top-level environment or harness module and drives all the chip I/O interfaces using target-RTL or, if that is insufficient, bridges. Bridges are so named because they span the host and target domains. The *target side* of a bridge consists of a Chisel module, generally a black box, whose instance is annotated with a bridge annotation, and whose ports are annotated to specify how the interface should be divided into token streams. The *host side* of a bridge is a unit (a primitive LI-BDN) that implements that blackbox module. Just like an endpoint, the host-side unit consists of a module (*BridgeModule*) and a driver

(*BridgeDriver*). Finally, to enable optimizations, users identify specific structures in the target by annotating them. As in MIDAS, additional compiler-side features are controlled using a parameter object which enables instrumentation passes and configures host-platform support. Additional user-supplied host and target FIRRTL transforms are provided with this parameters instance.

Since it does not require access to unserializable Scala types like MIDAS, Golden Gate can be invoked as a standalone application on an elaborated target. This permits running the compiler on a target elaborated in a different environment that may include a different versions of Chisel, FIRRTL, and Rocket Chip than those the compiler itself depends on. We give an example invocation of Golden Gate in Listing 4.1.

```
sbt runMain midas.stage.GoldenGateMain \
    -o <output filename> \
    -td <output directory> \
    -i  <input FIRRTL filename> \
    -faf <input FIRRTL annotations file> \
    -ggcp <Golden Gate parameters scala package> \
    -ggcs <Golden Gate parameters scala class string> \
    -E verilog
```

Listing 4.1: An example command-line invocation of Golden Gate.

4.3 Updated Compiler Flow

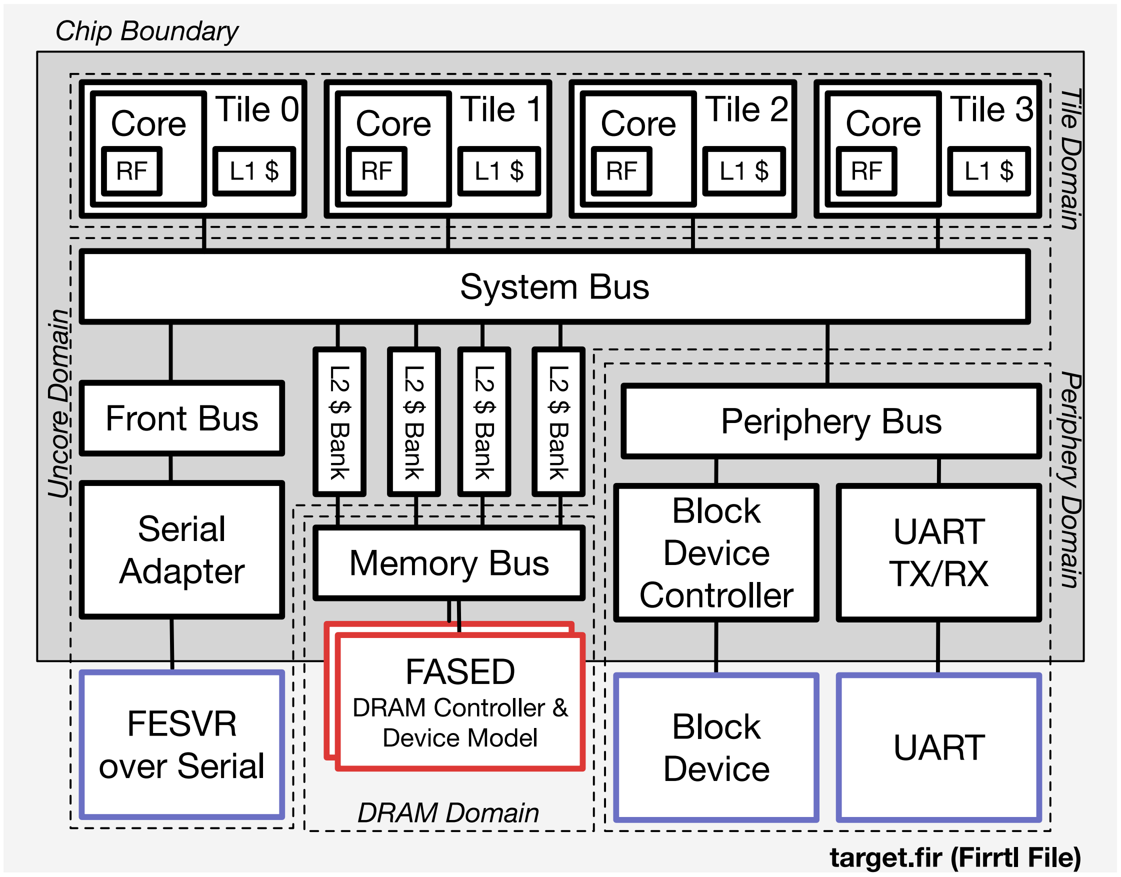
We show a depiction of Golden Gate's compilation flow and how it modifies an illustrative toy design in Figure 4.2. We omit debug-synthesis transforms, and user-provided transformations for brevity. Golden Gate tranformations, which are statically ordered in `MidasTransforms`[2], can be roughly[3] divided into three phases:

1. **Target Transformation.** Here the module hierachy is mutated in preperation for host decoupling. Bridges are extracted and unsynthesizable debugging constructs are replaced and bound to a bridge interface. Primitives identified for optimization are wrapped in modules and labelled with additional optimization annotations. These modules are then promoted to the top of the design hierarchy. All top-level module instances correspond to units in the eventual simulator and top-level connectivity between these modules define edges in the graph.

2. **Simulator Synthesis.** Here top-level modules are transformed into, or replaced with, primitive LI-BDN units. On completion, the output circuit consists of a series of unconnected unit instances. Connectivity between units is captured in channel annotations between top-level I/O.

[2]See sim/midas/src/main/scala/midas/passes/MidasTransforms.scala

[3]Note, these terms are not currently enshrined in the codebase.



Optimization Annotations

```
RAMOptAnnotation(tile0.core.rf),
RAMOptAnnotation(tile1.core.rf),
RAMOptAnnotation(tile2.core.rf),
RAMOptAnnotation(tile3.core.rf),

MultithreadingAnnotation(tile0),
MultithreadingAnnotation(tile0),
MultithreadingAnnotation(tile0),
MultithreadingAnnotation(tile0),
```

Channel Annotations
<Label Bridge Interfaces>

Bridge Annotations

```
BridgeAnnotation(
  target = blockdev,
  bridgeClass = BlockDeviceBridge),

BridgeAnnotation(
  target = uart,
  bridgeClass = UARTBridge),

BridgeAnnotation(
  target = serial,
  bridgeClass = SerialBridge),

BridgeAnnotation(
  target = fased0,
  bridgeClass = FASEDMemoryTimingModel),
```

target.anno.json (Annotations File)

Figure 4.1: A typical Chipyard SoC passed to Golden Gate (top), and its associated Golden Gate annotations (bottom). Clock domains are shown for reference in later chapters.

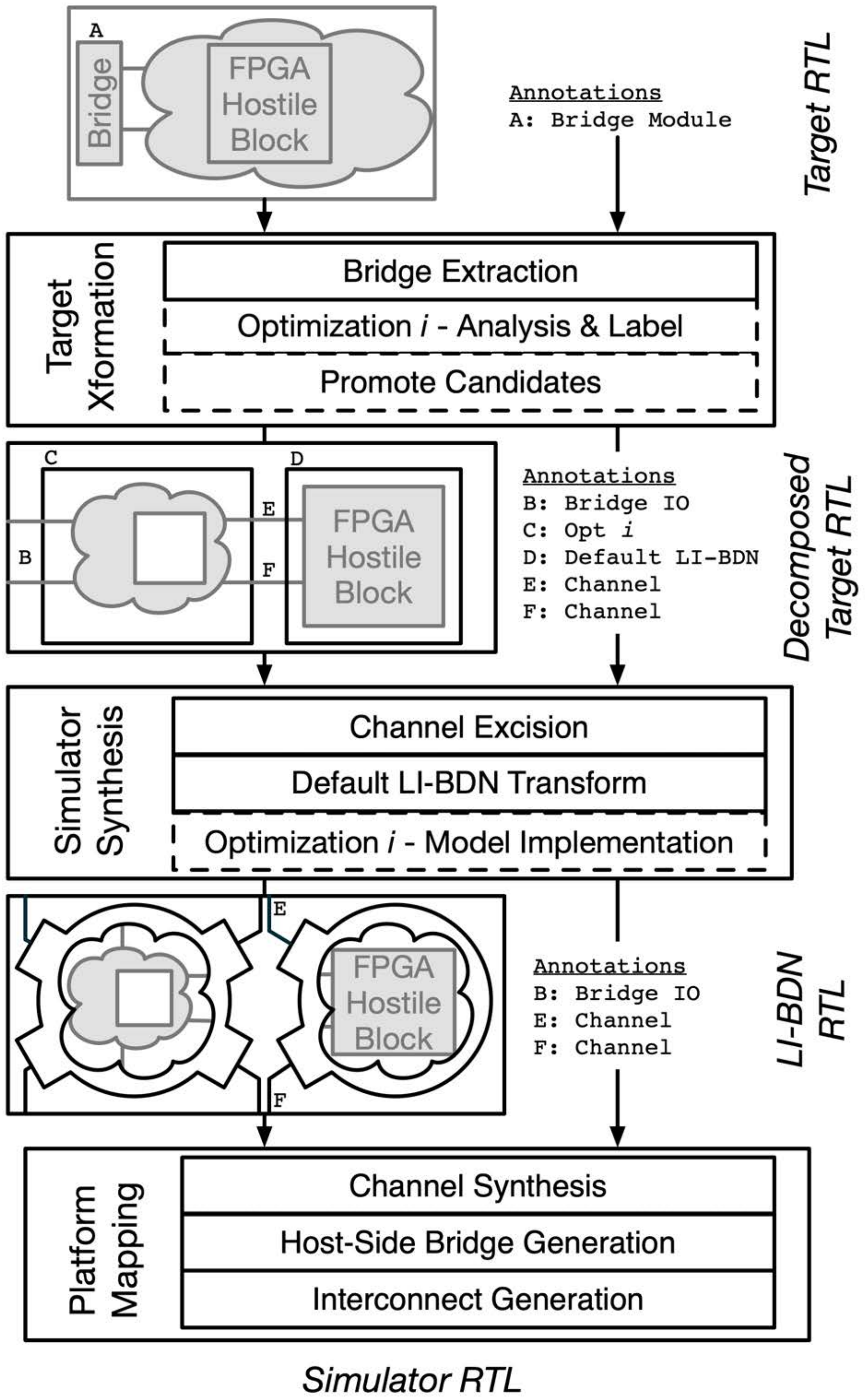


Figure 4.2: Core phases of the Golden Gate compiler and a high-level depiction of the transformations made to the underlying target and its annotations. The default LI-BDN transform is the FAME1 transform.

3. **Platform Mapping.** This consists of the Simulation Mapping and Platform Mapping passes of MIDAS. A Chisel-generated wrapper includes all channel implementations and elaborated host-side bridge modules. The resulting output circuit is compatible with the same FPGA projects as MIDAS-generated simulators and plugs directly into FireSim's existing build flow.

4.3.1 Annotations

Golden Gate relies on FIRRTL's annotation system to communicate information about the circuit from transform-to-transform instead of an auxiliary datastruture that persists across transformations. The main motivation for doing so was to make transforms in the compiler robust against running lowering or optimization transformations between core passes. The FIRRTL compiler framework provides callbacks for defining how an annotation should be updated when the circuit structures it targets are changed (a process known as *renaming*). We describe the key annotations here:

- **Bridge Annotations** (`BridgeAnnotation`) label module instances in the target for which a custom unit will be generated. In addition to labelling a target module instance, they carry a fully qualified class name of the Bridge Module to be generated and its optional constructor parameter. Finally, the annotation enumerates the names of all of its channel annotations.

- **Connection Annotations** (`FAMEChannelConnectionAnnotation`), or *channel* annotations, label connections between modules that correspond to units and bridges of the simulator, and carry the information required to generate a channel to replace these connections. A channel annotation consists of a list of sources (`sources`), I/O fields on the instance that drives the connection (i.e., the instance sources the tokens), and sinks (`sinks`), I/O fields on an instance receiving those connections. Channels carry additional metadata, including a globally unique string identifier (`globalName`), as well as a case class (`chInfo`) that captures target properties of connection and provides implementation hints. Notably, `chInfo` has subclasses to indicate whether the channel is latency insensitive and should be implemented with a queue-type channel, or is a pipe-type channel.

- **Port Annotations** (`FAMEChannelPortAnnotation`) label groups of I/Os in a module as corresponding to a channel source or sink. The list of sources or sinks in connection annotations (which point at instances of the module) must agree with the port annotation on the module itself. Port annotations are primarily consumed by transformation passes and provide the simplest means to look up how the I/Os of the module should be divided into channels.

- **Model Annotations** (`FAMEModelAnnotation`) serve a similar function to BridgeAnnotations, in that they label a module as being a unit. Unlike bridges, instances of the target module are not extracted from the design hierarchy but instead are promoted to the top-most level of the design hierarchy so that they can later be transformed by the compiler in Simulator Synthesis.

At the start of compilation, bridge annotations are present, with connection annotations labelling each bridge's interfaces. Strictly speaking, this is an abuse of the abstraction presented by the annotation (described above), which is fully realized only once bridges are extracted. Finally, model annotations may also present alongside specific optimization annotations.

4.4 Target Transformation

The primary function of Target Transformation is to coerce the design's module hierarchy into a form that matches the eventual simulator topology: top-level instances, which wrap optimization candidates, map to unit instances (nodes) and top-level connections map to channels (edges). Throughout this process, no host-time simulator constructs are introduced: the RTL still directly represents the chip, albeit with a different module hierarchy. Target Transformation consists of all transformations before top-level connectivity is removed (this occurs in `ChannelExcision`).

Target Transformation commences with an initial lowering and optimization of the input FIRRTL, after which the input circuit is checked for Golden Gate compatibility (namely, the circuit presents no top-level I/O; see `EnsureNoTargetIO`). Next, Golden Gate runs assertion and `printf` synthesis. Under-the-hood these do not fundamentally differ from what was described in our DESSERT publication [57], however synthesized debug primitives are now bound to bridges (more on this later).

Now Golden Gate finds all modules annotated with a bridge annotation and extracts them (`BridgeExtraction`). Connections to instances of a bridge are replaced with connections to newly added top-level I/O. Note that a single bridge module with n instances will produce n sets of ports at the top-level. The bridge and connection annotations are "promoted" to point at these new top-level interfaces. Bridge annotations are then replaced with a form of bridge annotation that targets top-level I/O (`BridgeIOAnnotation`) instead of a module but are otherwise identical to the user-emitted form. Finally, connection annotations are assigned new `globalName`s to ensure they are unique; corresponding bridge annotations are updated to reflect naming changes.

Next, Golden Gate prepares for its optimization-focused hierarchy manipulations by adding a wrapper module, the *FAME wrapper*, around the existing design (`WrapTop`). Once optimization candidates have been promoted, the previous top-level module will become the hub-unit of the simulator.

It is at this point we envisioned custom optimization analyses would be run find and label optimization opportunities. Currently, the multi-ported RAM analysis is the only example of this in the code base (see `LabelSRAMModels`). This pass finds FIRRTL memories that have been annotated as safe to optimize, wraps them in a module, and labels them with a model annotation.

Now Golden Gate finds all module instances labeled with a model annotation, and promotes them to the FAME wrapper. This process is nearly identical to bridge extraction, but the module is not completely removed. Just as in bridge extraction, one annotated module will produce as many top-level instances as there were instances of the module in the design hierarchy. At this point the connectivity within the wrapper reflects the star-topology of the eventual simulator: the original source module is the hub and all promoted modules and bridge interfaces connect directly to it.

With the module hierarchy manipulations finished, Golden Gate completes Target Transformation by fleshing out missing annotations (`FAMEDefaults`). This primarily consists of labelling all inter-model connectivity with connection annotations. This pass makes no attempt to group connections into a single channel: each ground-type connection between the hub and a spoke gets its own annotation and thus will receive an independent channel implementation.

4.5 Simulator Synthesis

Simulator Synthesis is so named because it begins introducing host-time constructs into the circuit, most notably with the execution of the default FAME1 transformation. Simulator Synthesis begins by removing all inter-model connectivity and replacing it with top-level I/O (`ChannelExcision`). Connection annotations are promoted to point at these new interfaces. Only now are port annotations added (`InferModelPorts`), since connections between nodes (both bridges and models) now uniformly target top-level I/O. This transformation ensures that connection annotations on instances of the same module agree as to how ports correspond to channels. Since `FAMEDefaults` makes no attempt to combine leaf channels into aggregate ones, this mainly serves as a consistency check on bridge-bound channels and ensures no connection annotations have been lost.

The bulk of the complexity in Simulator Synthesis is contained in its default FAME1 transform (`FAMETransform`). It has two fundamental tasks: first, it transforms all top-level modules, including those that will be optimized later, into units using a default, wrapper-based LI-BDN transform (described previously and shown in Figure 3.4). The transform groups ports into new decoupled interfaces by consuming the port annotations for the module under transformation. References to target ports are replaced with references to the payloads of these decoupled channel ports. Valid and ready signals are used to build out the control circuitry shown in Figure 3.4, however, this logic is appended to the existing module (to avoid introducing another level into the module hierarchy). Finally, all references to the target clock are replaced with references to the output of a clock buffer which is responsible for driving a selectively gated target clock. The clock buffer itself is left as a black box that will be implemented later for the desired host FPGA. The second major function of the FAME transform is to rewire the FAME wrapper by replacing existing I/O and connectivity with decoupled equivalents. For example, a boolean driven by a model is replaced with a three-bit interface: the valid and boolean payload signals are driven by the model, and the ready signal is driven by the I/O interface in the FAME wrapper. Connection annotations are updated to point at the payloads of these channels.

At this point, Golden Gate runs optimization transformations to replace the default unit implementations. Since the default FAME transformation has already decoupled the interfaces of the reference SSM, these transformations simply replace the module body of the existing primitive LI-BDN units. Connectivity in the FAME wrapper remains unchanged. Simulator Synthesis concludes with some fix-up transformations which provide a default clock buffer implementation (`DefineAbstractClockGate`) compatible with RTL simulators, and wires up the host-clock and reset to all units in the FAME wrapper (`ConnectHostClock`).

4.6 Platform Mapping

In Golden Gate, platform mapping is responsible for implementing all channels (previously known as simulation mapping in MIDAS) and elaborating all remaining simulation collateral, such as bridges and resource interconnect (previously platform mapping in MIDAS). This happens in single Chisel invocation, and produces three layers of wrapper modules, shown in Figure 4.3.

4.6.1 Channel Synthesis

The first wrapper layer (`SimWrapper`) directly interfaces with the transformed RTL and instantiates channel implementations. Elaboration of this module is driven by two inputs: the user-defined parameters instance which, as in MIDAS, specifies properties about the host-platform. and the annotations, notably channel annotations, labelling the now transformed target. Each channel annotation will correspond with a channel implementation, however how that channel is bound depends on the `sources` and `sinks` enumerated by the annotation. Channel annotations that possess both `sources` and `sinks` connect two models in the transformed RTL: these are *loopback* channels that drive an input and an output interface on the transformed target. Conversely, annotations that possess an empty `sources` or `sinks` parameter are sunk or driven by a bridge module, respectively. In these cases, the channel implementation has its dequeue or enqueue side interface exposed to the next wrapper layer of the module hierarchy.

In all cases, the type of channel generated is parameterized by the `channelInfo` field of the channel annotation. Target-decoupled channels, as in MIDAS, always instantiate a model of a fully decoupled, two-deep queue. Pipe channels have a configurable latency. Default bridge implementations always emit pipe-type channel annotations with a single-cycle latency to improve FMR, and to maintain the performance characterisitics of legacy MIDAS simulators. When no additional models are extracted (i.e., the simulator consists only of the hub unit and bridges) the expected FMR is identical to MIDAS. When additional

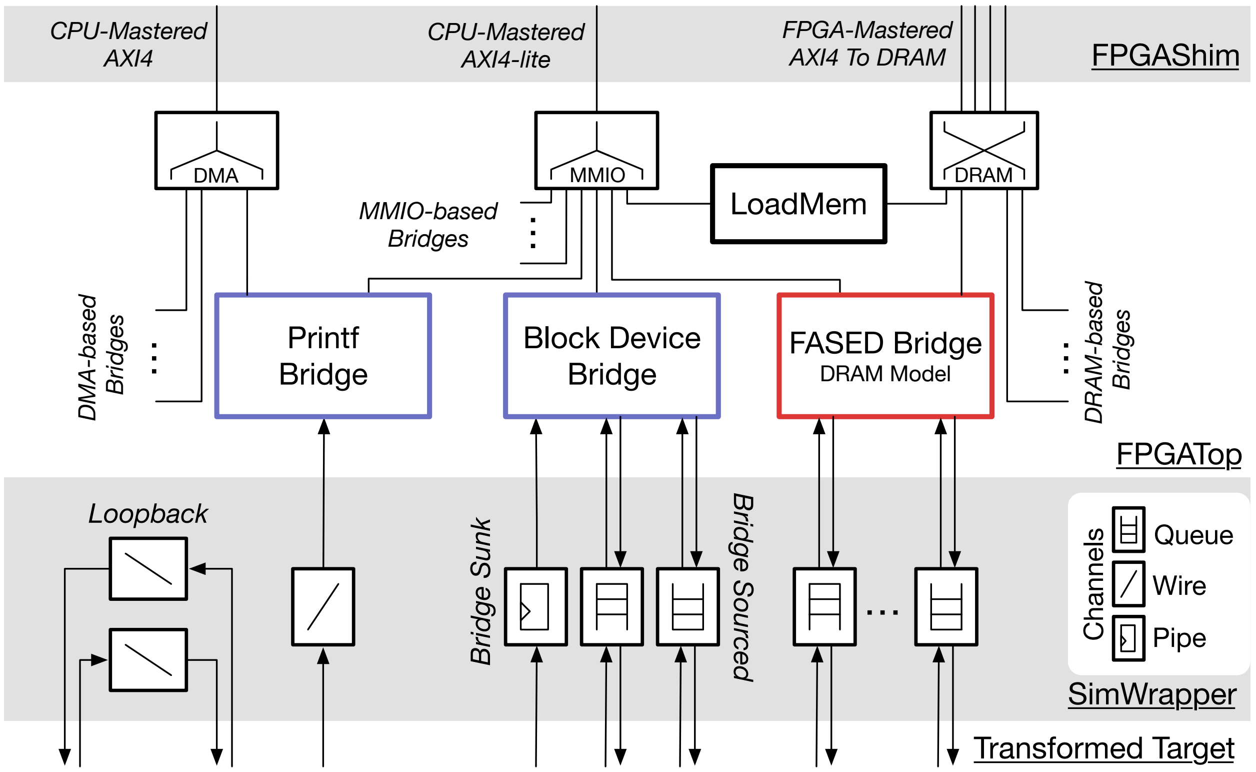


Figure 4.3: The wrapper circuit generated during platform mapping, with a handful of bridges shown for illustration. The host interfaces at the top (in `FPGAShim`) match those used in MIDAS. Interconnect between host interfaces and bridges initially was identical to that in MIDAS, but has since been augmented. Loopback channels, not present in MIDAS, connect two units in the transformed target.

units are extracted, they are always directly wired to the hub, since the compiler currently cannot find and extract registers (that is to say, `FAMEDefaults` labels these channels as wire-type). While it is possible to build simulators with unity FMR if an optimized model's outputs can runahead, at time of writing, we have no such implementations. Therefore, in practise all multi-model simulators execute with an FMR of at least two.

4.6.2 Bridge Instantiation

The next level of the module hierarchy (`FPGATop`) corresponds to the wrapper circuit generated in MIDAS's platform mapping. I/O on this wrapper module consist of AXI4 interfaces to drive host-DRAM memory systems, and to support CPU-driven MMIO and DMA. Instead of using an endpoint map, Golden Gate iterates through each bridge annotation and reflexively invokes the constructor for the requested `BridgeModule`. I/O between the elaborated bridge instance and the simulation wrapper are connected by looking up the corresponding channels via their `globalName` fields. Resource-interconnect generation is identical to MIDAS's, and for the same input design, the emitted header is the same. All MIDAS endpoints were ported to use the new Bridge system without changing existing host software.

4.6.3 Platform Wrapper

The last level of wrapper module is an FPGA-specific shim layer. The user selects their desired shim through the parameters instance. This serves as an opportunity to introduce FPGA-specific hardware and modify interface names to link into an FPGA shell project. We note that this top-level module could instantiate all of the IP required to define a complete FPGA project (as in SiFive's FPGA shells repository), but none of our shims do so presently. The class for this wrapper module can be defined outside of Golden Gate and provided by a user wishing to support a non-standard FPGA.

4.7 ICCAD 2019 Golden Gate Publication

Armed with the machinery to selectively extract and reimplement FPGA-hostile components of the target design, in our ICCAD2019 publication [70] we set about optimizing multi-ported RAMs (we previously introduced this example in Section 3.1). ASIC multi-ported RAMs are a classic culprit for poor resource utilization in FPGA prototypes, as they cannot be trivially implemented in BRAM and are instead decomposed into LUTs and registers [105]. While using double-pumping, BRAM duplication, or FPGA-optimized microarchitectures [64] can help, here we used Golden Gate to replace the SRAM with a multi-cycle unit to further reduce resource utilization. This optimization implements a target memory possessing M asynchronous read ports and N write ports with a single time-multiplexed dual-ported BRAM. In contrast to the static time-multiplexing schemes described by Pellauer et al. [80] and Dwiel et al. [33], our optimization can decouple arbitrary target memories and replace them with an optimized model without any constraints on the structure or timing of the target design.

In order to rigorously verify optimizations, in our ICCAD publication we also introduced a bounded-model checking flow, called Latency-Insensitive Model Equivalence (LIME). Hosted in UCLID5 [89], LIME formally verifies that optimized models are valid LI-BDNs implementations of their reference RTL. Our manipulations of the module hierarchy were in direct service of this goal, as any extracted RTL module could trivially serve as the reference SSM for a verification flow. In our ICCAD paper, we used LIME to verify only our RAM optimization, however, there is nothing that precludes applying LIME to *all* reference-unit pairs in a simulator. This verification flow could be run in parallel to FPGA compilation, providing a multi-hour window to catch potential bugs before a simulator is deployed to FPGA.

The LIME model checking flow and RAM optimization first described in our ICCAD2019 paper are primarily contributions of Albert Magyar and are described at length in Chapters 6 and 7 of his dissertation [71].

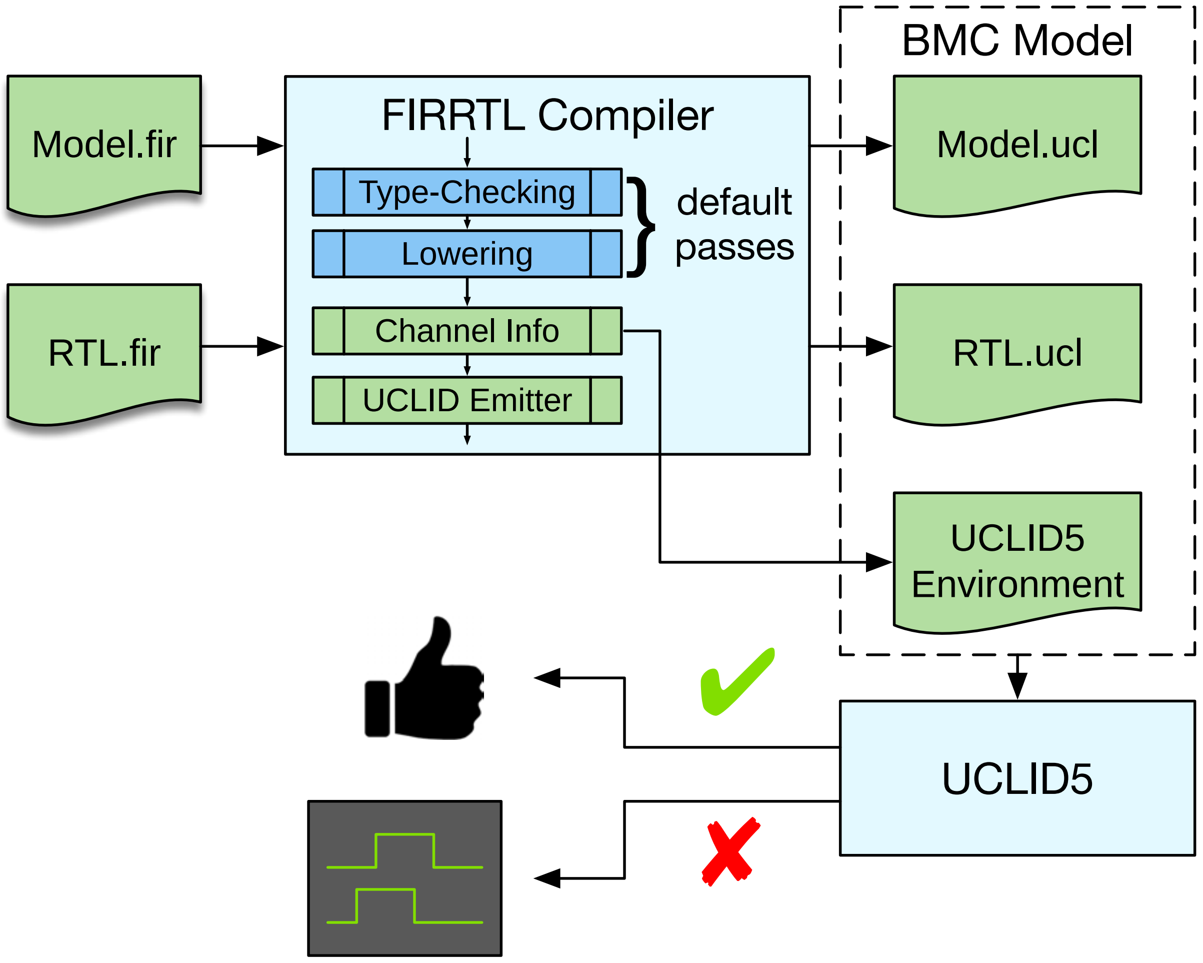


Figure 4.4: The LIME model checking flow. In LIME, FIRRTL for a unit and its reference SSM are lowered and emitted as UCLID5 representations. A Python-based environment generator (not shown), consumes channel information emitted by the compiler to generate test environments for PI, GC, and NED. If the unit is not a valid LI-BDN, LIME produces a waveform counterexample that demonstrates how the property under test was violated.

Chapter 5

Support for Systems With Multiple Fixed-Frequency Clocks

Until now all Golden Gate and MIDAS-generated simulators modeled systems with a single clock domain. Anecdotally, the lack of support for simulating targets with more than a single clock has been the most common criticism of FireSim offered by potential users, many of whom turn to using a conventional FPGA prototype. The criticism is well justified: forcing the outer memory hierarchy to run at the same frequency of the cores results in a memory hierarchy that can sustain artificially high bandwidths at lower latencies.

To date, the only effort to validate FireSim performance against a silicon implementation was conducted by Lee and Waterman [67]. They compared SPEC2006 Integer results collected from the HiFive Unleashed [91] against an equivalent design running on FireSim. While the FireSim-collected SPEC score fell within 2% of the HiFive Unleashed's, the largest single-benchmark difference was 10.7% for `429.mcf`, a benchmark with infamously poor memory locality (its working set has been measured at 680 MB[44]). The FireSim variant differed in that it used non-standard UC Berkeley I/O devices (and backing FireSim Bridges) and a FASED memory timing model instead of an ASIC DRAM memory controller. However, the most pertinent difference between the two platforms is that the HiFive Unleashed core complex and memory hierarchy span three clock domains. Tiles run at the fastest frequency (1.5 GHz in their study), the uncore runs at half this frequency (750 MHz), and the DRAM memory system, capable of 2400 MT/s (resulting in a 600 MHz controller clock in the Unleashed), sits in its own clock domain behind an asynchronous crossing [92]. To model these domains without native support for multi-clock simulation, they resorted to adding additional buffering between the L2 and the inner caches, and scaling FASED latencies to match the times expected from a DRAM controller running at a slower frequency. Systems like the Freedom Unleashed, which do not actively scale frequencies after boot, are relatively common. For these systems, support for multiple statically defined clocks would suffice to resolve this performance discrepancy.

To implement this feature, a natural place to look for inspiration is in FPGA prototyping. There, supporting multiple clock domains is a relatively straightforward process in theory. When mapping an ASIC design to an FPGA, clock-generating circuits, like PLLs, are replaced with FPGA equivalents [4]. While the absolute frequencies used in the prototype will be considerably slower due to frequency limitations of the FPGA, the relative frequencies can be maintained and thus, latencies through an SoC cache hierarchy can be properly modeled (though, the aforementioned problem of modeling off-chip memory systems like DRAM remains). In practice, limited availability of clocking resources and restrictions in FPGA clock distribution can sometimes require non-trivial changes to the ASIC RTL. To ameliorate these challenges the *FPGA-Based Prototyping Methodology Manual* [4] suggests a number of Design-For-Prototyping techniques such as implementing only a subset of the clocking structures, sticking to conventional synchronous design techniques, and isolating clock generation and distribution structures in separate modules at the top-level of the design hierarchy.

Applying the prototyping approach to a decoupled simulator—generating multiple host-clocks whose relative frequencies match that of the target—is an abstraction-breaking change that conflates host and target concerns. In practice, optimized models and bridges are going to be resident in different target clock domains, and thus it will be necessary to independently gate the different host clocks. This destroys much of the merit of generating multiple clocks in the first place. Instead, we can derive multiple simulated clocks from a single host clock. This approach has some appealing implications:

- It does not require FPGA-specific clocking resources beyond clock buffers capable of gating the host clock. For smaller units where the fanout on the clock enable is small, even these can be avoided by directly adding a clock enable to all state elements in the domain. This makes it easier to port the same simulator to a different FPGA.
- It simplifies the implementation, since all simulator control logic is synchronous to the same clock.

One potential benefit of generating host clocks with different frequencies is that it has the potential to improve simulator throughput (3.1) by putting faster parts of the target in faster host clock domains (thus improving f_{fpga}). Intuitively, one would expect that a faster target clock domain should close timing at higher frequency than a slower one. While this is generally true, the ratio of critical-path delays between clock domains can differ substantially from an ASIC implementation, because delay through ASIC elements do not scale uniformly across all structures when mapped to an FPGA [63, 105]. We do not rule out using multiple host clocks in the future, rather, we argue that host clock frequencies should be selected based on simulator critical paths specifically to improve simulator throughput, not as a means to enable simulation of multiple clocks in the target.

Using a single host clock still enables a variety of implementation styles. Our initial prototypes revolved around modeling clock-domain crossings in channels. For example, one could model a two-to-one crossing from a fast clock domain to a slow clock domain by dropping every second token or, if in the reverse direction, duplicating every token. An early prototype of this approach can be found in FireSim version 1.4, which permitted users to model a clock division in the crossing between the hub and an endpoint. Huang et al. [49] used this to simulate targets with a DRAM memory system running at one third the rate of the rest of the simulator. To apply this technique across the target and not simply between bridges, we considered using Golden Gate's hierarchy manipulations to divide the target design into synchronous islands. Each of these islands would become separate units transformed with an unmodified FAME transform. In simulation mapping, Golden Gate would synthesize wire-type CDC channels between these islands. Note that the clock-domain crossing present in target still exists, but it is split into two synchronous halves across the units. Clock frequency information would be baked into CDC channels during channel synthesis. We began a prototype implementation of this approach (we sketched out FIRRTL transformations to perform this partitioning and designed the channels) before reconsidering. This approach introduced considerable structural changes to the design's module hierarchy, making the simulator more difficult to debug, and complicating a reimplementation of Strober-style state snapshotting. Secondly, it introduced non-trivial amounts of queuing when cuts between clock domains spanned large sets of signals. Perhaps most importantly, it was considerably more complex than the solution we ultimately selected.

5.1 Implementation

Our approach instead was to modify the hub unit to simulate multiple clock domains *in situ*. The hub unit instantiates separate clock buffers, specifically Xilinx BUFGCE primitives, for each clock domain, and selectively fires clocks and channels based on the set of clock edges it is scheduled to simulate. A single *clock bridge*, generates the clock schedule as a token stream of bit-vectors indicating which clocks are scheduled to fire in a given simulator timestep.

5.1.1 Simplifying Assumptions

To expedite the implementation of our initial prototype, we made the following assumptions:

1. The behavior of all clocks can be statically deduced. To further restrict this, we also mandate that all clocks are rationally related.
2. All clocks in the target must be sourced from the singleton clock bridge. Specifically, when in FIRRTL's low form, all clock-type members of FIRRTL statements and expressions must be driven by the clock bridge exclusively through a sequence of `Connect` statements.
3. All clock sinks must be positive-edge triggered. While Chisel has no native support for negative-edge-triggered or level-sensitive state elements, the user must avoid using these structures in black-box Verilog.
4. It must be legal to replace asynchronous resets with synchronous ones. Unlike in an ASIC implementation, the designer can exploit the fact that all clocks in the target

will come up at time zero to avoid using asynchronous resets where typically necessary. The implementation outlined in this chapter has no mechanism to separately handle launching asynchronous reset edges. This restriction will be relaxed in Chapter 6.

5. All extracted models and bridges must be synchronous. This permits all existing resource optimizations to remain unchanged, and simplifies bridge design.

All of these assumptions were implicitly made about the target in older versions of FireSim. The only difference lies in how the target clock is sourced: instead of using an explicit bridge, the target clock was driven by a input (this was the only legal I/O permitted on the module) on the source FIRRTL.

Strictly speaking, a target with more than one clock cannot be an SSM, and so generated multi-clock simulators will no longer be LI-BDNs. However, extracted models will remain primitive LI-BDNs with reference SSMs. We suspect that multi-clock RTL that conforms to the restrictions above can be cast as an SSM synchronous to a single fast clock whose equivalent slow clocks have been replaced with selectively enabled state elements. We leave an extension of LI-BDN to support targets the conform to the assumptions above as future work.

5.1.2 Annotation Modifications

Where previously all channels were implicitly synchronous to the sole target clock, now channels bound for different units may be synchronous to different clocks. To support this, we extended channel annotations to carry a reference to a clock in the hub model, so that the FAME transform may associate the correct subset of channels with each clock domain. Bridge-emitted channel annotations indicate their clock at instantiation time, whereas compiler-extracted models have their clock inferred (more on this later). Additionally, we introduced a new clock-type channel field (`TargetClockChannel`) to label the singleton clock channel in the simulator. This must be handled specially by the FAME transform and channel synthesis.

5.1.3 Modified FAME Transform

The bulk of the complexity in supporting multiple clock domains is contained in modified FAME transform. We show the resulting hub unit implementation in Figure 5.1. Unlike in the single-clock implementation, the wrapper circuit must selectively operate on a subset of channels based on which clocks are scheduled to fire. Output state machines (shown in blue) remain mostly unchanged but are now selectively reset based on the scheduled clock. We added input FSMs (shown in green), essentially a pipeline stage, to help cut the fanout delay from the clock token port to input channel dequeue. These too are selectively reset.

Clock scheduling is managed by a two-cycle control path (shown in orange). In stage one, input FSMs (in green) are reset so that they may dequeue a new input token in the

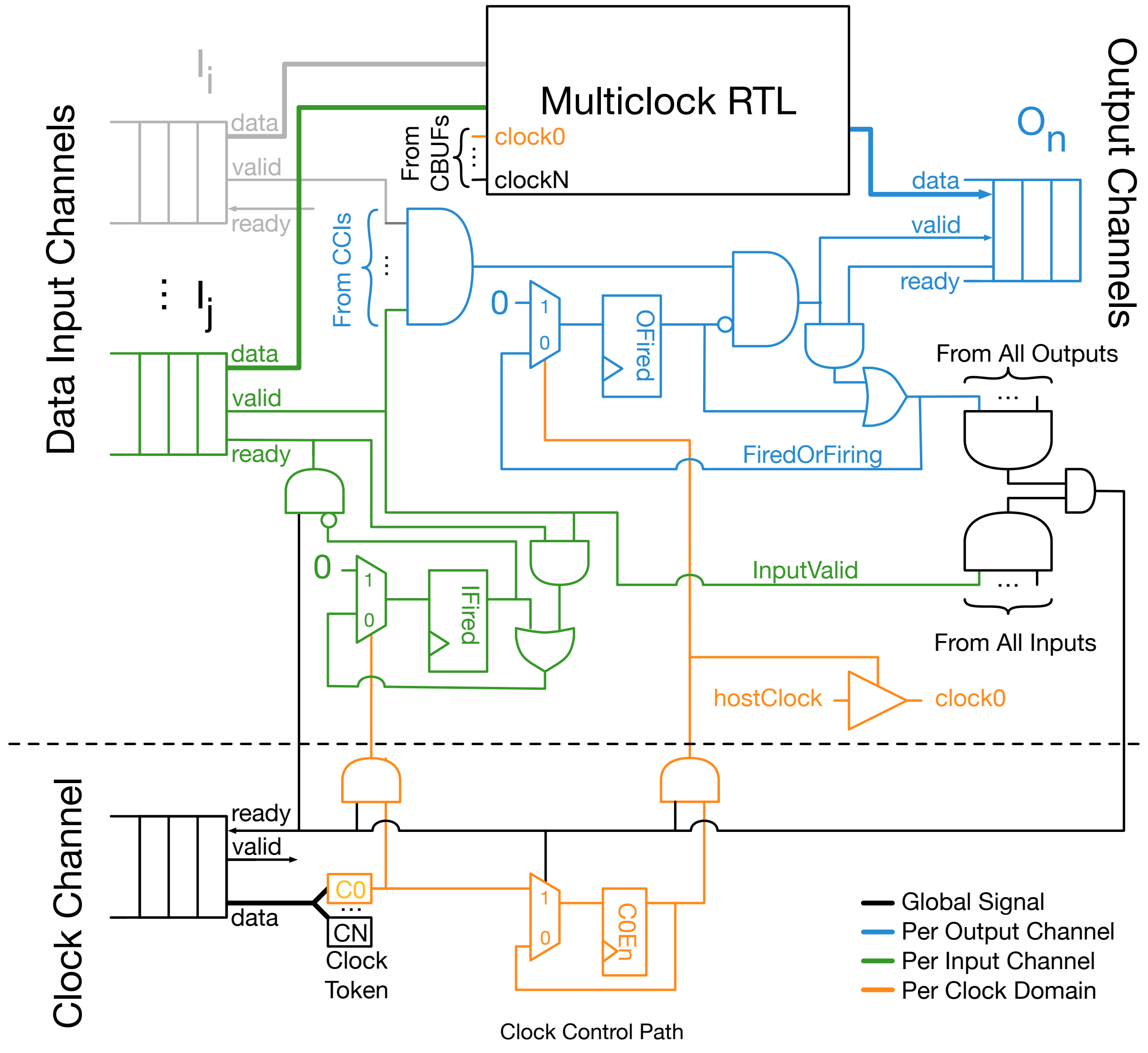


Figure 5.1: A wrapper-module-based conversion of multi-clock target RTL into a unit.

second stage. In stage two, output FSMs are reset and a pulse is launched through the target clock by enabling a dedicated clock buffer for that domain. The pipeline advances under largely the same firing condition as the synchronous wrapper: all channel inputs must be valid, including the clock channel, and all outputs must have enqueued or be enqueuing. Since output channels for clock domains not scheduled to fire in stage two are not reset, their `fired` registers remain set and cannot stall the pipeline.

Initially all output FSMs are reset to zero, and the clock pipeline register marked invalid (no clock is scheduled to fire). This permits all combinational paths across all clock domains resolve based on the initial input token values and target state. As previously discussed, all target clocks are gated under host reset such that BRAM and register state initialized

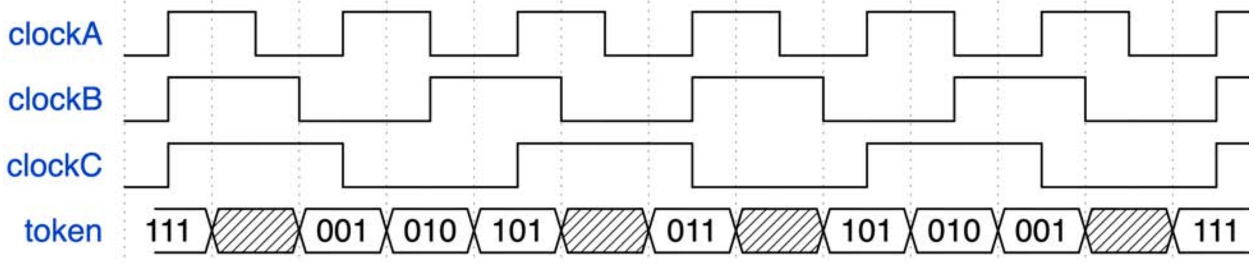


Figure 5.2: An example encoding of three rationally related clocks (clocks B and C have periods 1.5 and 2 times longer than clock A) into a clock token stream. Here there is a recurrence after eight tokens, with the fastest clock active in all but two tokens.

during FPGA programming cannot spuriously change as the FPGA comes out of reset.

5.1.4 Clock Bridge

The clock bridge is responsible for determining the clock schedule by generating an infinite token stream of n-wide bit-vectors, where n is the number of clock domains in the target. We show an example of this encoding for three rationally related clocks in Figure 5.2. Each clock token corresponds to a simulator *timestep*. Any clock with a positive edge in a given timestep will have its bit set in the corresponding clock token.

Per our earlier assumption, the clock bridge implementation included in FireSim 1.10 accepts clock specifications that define the frequency of all clocks as a rational multiple of the frequency of the zeroth clock[1]. To model a clock that is not rationally related (e.g., a periphery clock that generated by a secondary PLL or is sourced from off-chip), the user should select a rational multiple that would best match the desired frequency of the clock. During elaboration, the clock bridge determines a virtual fast clock from which all output clocks can be derived via an integer division (in other words, it finds a clocks whose frequency is the least common multiple of the frequency of the requested clocks). Then for each output clock, it generates a series of down counters (`timeToNextEdge`) which are systematically reset to the division required to generate that output clock.

To generate an output token, the bridge does a min-reduction of `timeToNextEdge` across all clocks, and broadcasts the result. All clocks whose `timeToNextEdge` matches the minimum are scheduled clocks: their bit is set in the output token, and their `timeToNextEdge` register is reset to their division. Unscheduled clocks subtract the broadcasted time delta from their registers, simulating the advance of time. This implementation is capable of producing a clock token with at least one bit set every host cycle.

[1]Another reasonable approach would be to specify the periods of all clocks in an agreed upon timebase. We note that in this case all clocks are rationally related to a fast clock whose period equals the resolution of the timebase.

5.1.5 Other Compiler & Bridge Modifications

In our and our users' experience, bridges are difficult to write as they force the developer to reason about both host and target-time considerations. To prevent further exacerbating this problem, we elected to forbid bridges and extracted models from having channels in multiple clock domains. This had three primary implications:

1. User-instantiated bridges must explicitly indicate the clock to which they are synchronous. Bridge-emitted channel annotations contain references to this clock. This is a relatively trivial change, but one that affects user facing code.

2. Debug synthesis passes must emit multiple bridges. For example, assertion synthesis must generate a new bridge for each clock domain in which there is at least one assertion. To do so, we built FIRRTL analyses to find source clocks by walking clock connectivity (`FindClockSources`), and to group and wire nodes (like an assertion condition) to the top-level based on each nodes source clock(`BridgeTopWiring`).

3. To populate extracted-model channels with references to clocks, we introduced a transform (`FindDefaultClocks`, run before `ChannelExcision`) that analyzes top-level clock connectivity between extracted models and the hub. Any stateful module once extracted will have connection between a new clock output port on the hub module and a sink port on the model. By finding these connections, models can be labeled with a clock reference on the hub model and their channels annotations can be subsequently updated.

5.1.6 The Base Clock

In a target system with a single global clock it is natural to specify time in cycles of that clock. Bridges can easily keep time with a counter that tracks the number of times it has fired. Introducing multiple clocks complicates this: specifications of time must either be made explicit, or specified in clock cycles of some agreed-upon common clock. In either case, these specified times cannot be compared against a bridge-local cycle counter without the local clock's frequency, or its relative frequency to the agreed-upon common clock. This is problematic because many of FireSim's features use specifications of global time. For example, instrumentation bridges (e.g., assertion and print bridges) accept runtime arguments that specify windows of time over which they should be enabled. Since these features are implemented over multiple bridges in multi-clock targets, clock domain information must be provided to each bridge such that they can translate time specifications into local cycle counts.

Here we exploited our assumption that all clocks are rationally related and, to provide a user experience similar to the existing one, we elected to specify times in cycles of the zeroth clock generated by the bridge. We refer to this clock as the *base clock*. In Chipyard 1.4, the clock bridge is configured to make the fastest clock in the system its base clock, which in practice, always drives the cores of the design.

To propagate information about a bridge's clock domain to its host-side components, we added an analysis pass (`TargetClockAnalysis`, which runs during target transformation) that determines the clock index for each bridge by walking the clock netlist back to the clock bridge's output port. Using that index, the pass looks up the ratio of the bridge's clock relative to the base clock. This is passed through the parameters instance to each bridge module during platform mapping. Bridge modules typically serialize this information to the simulation header so they can be used in the driver to recalculate times specified in base-clock cycles in terms of their local clock.

5.1.7 Rethinking Simulation Performance

In Section 3.1. we described simple equations that govern simulation performance. Under a system with multiple clocks, these need to be updated. Among our users FMR (Equation 3.1) is a popular abstraction, and sees frequent use in conversations regarding simulation performance. Perhaps, the most natural extension of FMR for multi-clock contexts is to define it in terms of the fastest target clock:

$$FMR_{fastest} = \frac{cycles_h}{cycles_{t,fastest}} \tag{5.1}$$

Henceforth, when referring to FMR in the context of multi-clock simulators we'll be using $FMR_{fastest}$ unless otherwise stated. Given our implementation, when all other clocks in the target can be expressed as an integer divisions of the fastest target clock (this target clock is the virtual fast clock), the simulator can execute with unity FMR. However, this is often not the case, especially when modeling device clocks. For example, in a two-clock system where the second clock has a frequency $\frac{2}{3}$ that of the base, the best-case FMR the simulator can achieve is $\frac{4}{3}$. This occurs since every other slow clock edge will not be coincident with a fast clock edge and thus will require a simulator timestep in which no fast clock edge is processed. We show this effect in the token stream illustrated in Figure 5.2).

An alternate measure of simulator efficiency, one that reveals the presence of host-time stalls independent of frequency selection, considers all host cycles in which the hub model is firing at least one target clock. We define *model activity ratio* (MAR):

$$MAR = \frac{cycles_h}{timesteps} \tag{5.2}$$

Where timesteps is the number of host cycles in which one or more target clock edges are launched, or in other words, the total number of clock tokens consumed by the hub model. Like FMR, a larger MAR corresponds to worse simulation performance, however, in the absence of other host-time stalls all Golden Gate simulators can run at unity MAR.

5.2 Case Study - SPEC CPU 2017 Integer Performance

Armed with the capacity to simulate multiple clock domains natively, we can quantify the performance difference between single clock designs and more realistic multi-clock designs. For this experiment we used otherwise standard Chipyard SoCs (their block diagrams match that shown in Figure 4.1) and modified the clock domain organization. For simplicity, in all cases we coupled the periphery and uncore domains. We considered three different configurations:

1. A completely synchronous configuration running at 1.5 GHz. FASED DRAM timings were scaled up by approximately 1.5 times to match a DDR3-2133 speedgrade.
2. A two-domain organization with the memory bus and backing DRAM subsystem sitting behind an asynchronous CDC. Here the memory bus runs at 1.0 GHz, and the rest of the system remains clocked at 1.5 GHz.
3. A three-domain organization that loosely matches the Freedom Unleashed where tiles run at 1.5 GHz and the uncore runs at 750 MHz with a low-latency rational CDC between them. The DRAM memory system remains unchanged from the previous organization.

For each of these three organizations, we studied SoCs based around two fundamentally different pipeline microarchitectures: Rocket [8], a 5-stage, in-order scalar core, and SonicBOOM [108], the latest iteration of UCB-BAR's out-of-order superscalar core. We used the preset "Large" core configurations for both microarchitectures: their parameters can be found in Table 5.1. Since BOOM can dynamically schedule around hazards like load misses, it should be able to drive more memory system bandwidth (it has 4 L1 data cache MSHRs) and tolerate additional latency through the memory system. We kept the uncore and DRAM memory system configurations, which consisted of a four-bank, 2 MiB L2 Cache and FASED-modeled, single-channel DDR3 memory system, fixed across the two designs. Note, we are not using FASED's optional last-level-cache model, but instead an ASIC-optimized design generated using SiFive's open-source cache generator [40].

We used the ccbench [41] `caches` microbenchmark to measure round-trip latencies latencies through the memory system under all three clock organizations. In Figure 5.3, we show the loop trip latency when doing a randomized pointer chase over different working set sizes running on BOOM (note, the pointer chase touches each cache line only once). Marginal latency introduced in the two-domain design over the synchronous design can be attributed to the addition of an asynchronous CDC plus the additional latency introduced by register stages now running relatively more slowly than the cores (these could not be trivially scaled by multiplying a configurable value). Marginal latency introduced in three domain configuration can be attributed solely to running the uncore clock at half rate.

Parameter	**Rocket**	**SonicBOOM**
Machine Width	1	5 issue, 3 commit
ROB Depth	N/A	96
L1 I$ Capacity	16 KiB	32 KiB
L1 I$ Associativity	4-way	8-way
L1 ITLB Capacity	32	32
L1 D$ Capacity	16 KiB	32 KiB
L1 D$ Associativity	4-way	8-way
L1 DTLB Capacity	32	16
L1 & L2 Block Size	64B	
L2 TLB Capacity	1024	
L2 Cache Capacity	2 MiB	
L2 Cache Associativity	8-way	
L2 Cache Banks	4	
System Bus Width	128b	
Memory Bus Width	256b	
DRAM Speedgrade	DDR3-2133 (14-14-14-47)	
Memory Access Scheduler	FR-FCFS	

Table 5.1: Microarchitectural parameters of the two targets under test. Note, that BOOM's L1 cache under the under the "Large" preset is twice as large as Rocket's (32 vs 16 KiB).

In this case study, we'll look at single-threaded performance of SPEC 2017 Integer Speed benchmark suite. This suite has many of the same benchmarks as its predecessor, the SPEC2006 suite used by Lee and Waterman in their VLSI evaluation of FireSim, however each of the benchmark's inputs are larger to better stress modern memory systems. To provide some insight into the suite's memory system characteristics, in Table 5.2 we report dynamic instruction counts and L1 cache misses-per-kilo-instruction (MPKI) running on a Rocket configuration with the same L1 cache organization as the ones we use in our study[2]. We cross-compiled SPEC using Speckle [43] and GCC version 9.2.0 with -O2 optimizations enabled. We ran SPEC on top of simple Buildroot Linux distributions that we assembled using FireMarshal [81]. Our system-software setup is reproducible without modification using FireSim version 1.11 and Chipyard 1.4, following the instructions for building SPEC2017 provided in our documentation.

In Figure 5.4 and Figure 5.5, we report SPEC runtimes as speedups relative to the most detailed three-domain configuration. BOOM designs insulated themselves better from the additional memory system latency brought about by running the outer memory systems at slower frequencies, with synchronous designs running only 1.09× faster versus 1.15× for Rocket based designs. While in Rocket's case, speedups correlate closely with L1 cache MPKI, BOOM's ability to dynamically schedule around these misses complicates the story.

[2]These figures were originally reported in our 2019 FASED publication.

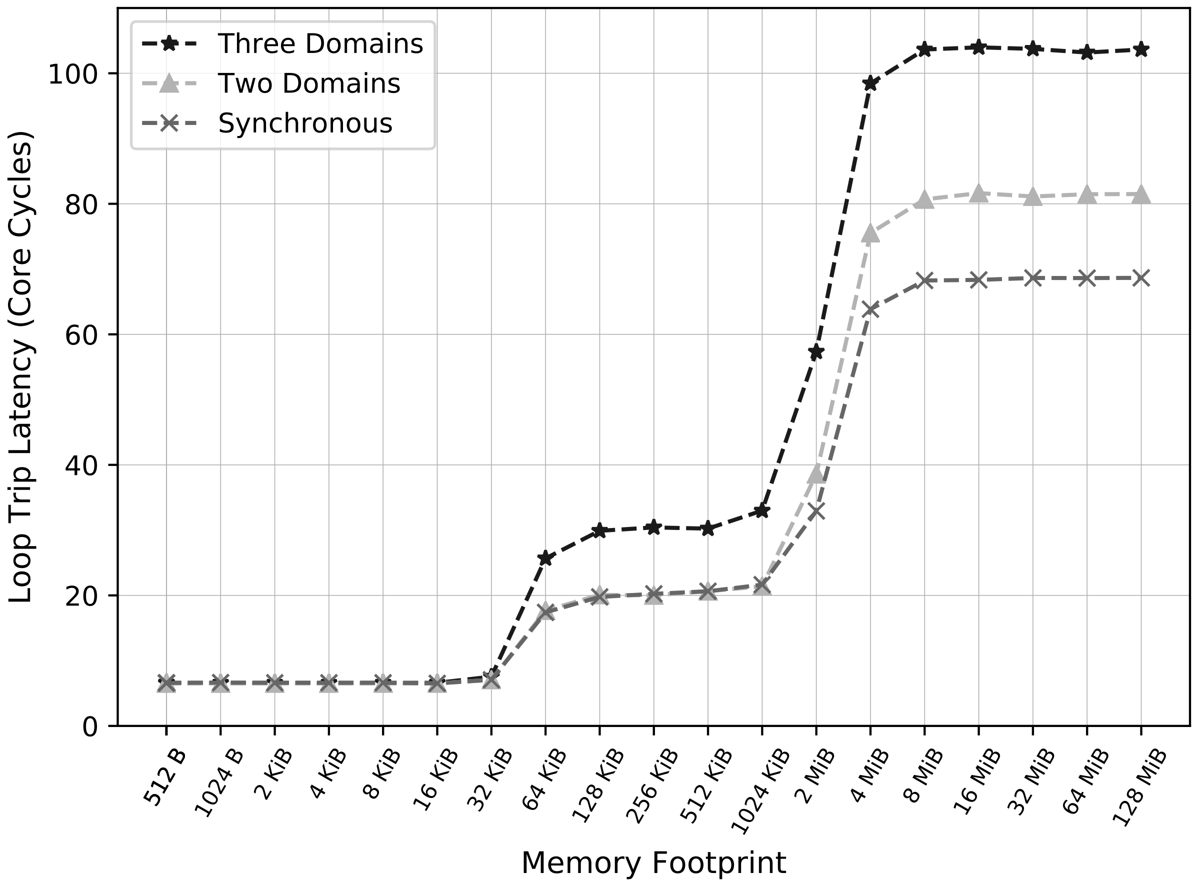


Figure 5.3: Execution latency, measured in core cycles, of one iteration of a randomized pointer chase for different array sizes (running on BOOM).

`605.mcf`, the benchmark with the highest frequency of L1 data cache misses[3], runs 1.37× faster on a synchronous Rocket design vs 1.20× on a BOOM design. Instead, BOOM sees its largest performance change on `602.gcc` (1.21× versus 1.26× for Rocket), which reveals that the core is able to find more instruction-level parallelism in `605.mcf` despite having less data locality than `602.gcc`.

From an emulation performance perspective, simulating a more realistic clock organization comes at the expense of increased runtime. As shown in Table 5.3, benchmarks take on average of 49.4% and 53.0% longer to simulate on the three-domain Rocket and Boom configurations, respectively. The drivers of this are twofold. First, since not all target clocks are integer divisions of the tile clock, simulation performance is bound to a best-case FMR of 4/3 (due to the effect we described in Section 5.1.7). This caps f_{emul} to 82.5 MHz and 45 MHz for the Rocket and BOOM-based designs, respectively. Second, the multi-clock designs

[3]This remains the case on the BOOM configuration's larger 32 KiB L1 cache.

Benchmarks	Insns (T)	D$ MPKI	I$ MPKI
600.perlbench_s	2.98	9.0	10.0
602.gcc_s	2.43	36.6	9.7
605.mcf_s	1.60	97.9	0.1
620.omnetpp_s	1.11	56.9	9.3
623.xalancbmk_s	1.21	62.9	7.9
625.x264_s	4.55	3.0	2.9
631.deepsjeng_s	2.51	8.7	15.4
641.leela_s	2.59	5.8	1.5
648.exchange2_s	3.24	0.0	0.1
657.xz_s	9.41	19.8	0.2

Table 5.2: Dynamic instruction counts and L1 MPKIs of SPEC2017 Integer Speed (single-threaded) benchmarks.

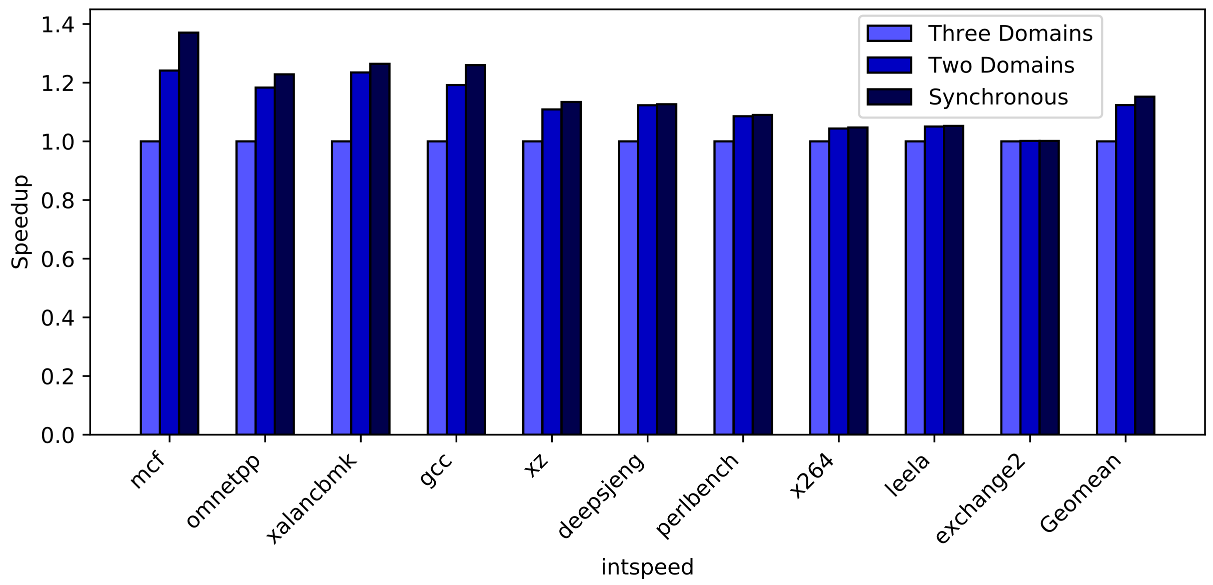


Figure 5.4: SPEC 2017 Integer Speed performance on Rocket. Speedups are relative to the three-domain clock organization. Geomean speedups for the two domain and synchronous configurations are 1.12× and 1.15× respectively.

are simply slower target-designs thus will take more target clock cycles to evaluate (this is captured in Figure 5.4 and Figure 5.5). These effects are offset modestly in multi-clock simulators by a reduced MAR: the host DRAM memory system has more time to serve FASED requests leading to fewer memory-system induced stalls. This is reflected in memory intensive benchmarks like `605.mcf`, where Rocket f_{emul} runs at 93% of the maximum 82.5 MHz (MAR = 1.08), whereas the synchronous design achieves only 79 % of 110 MHz

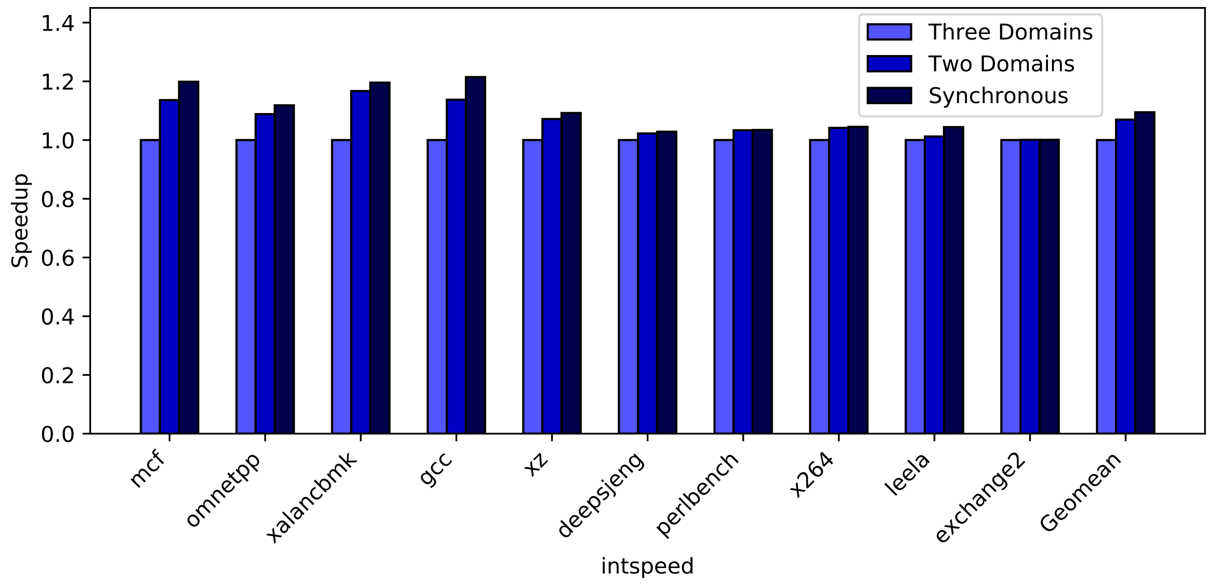


Figure 5.5: SPECint2017 Integer Speed performance on BOOM. Speedups are relative to the three-domain clock organization. Geomean speedups for the two domain and synchronous configurations are 1.07× and 1.09× respectively.

host frequency (MAR = 1.49). Conversely, the runtime of benchmarks like `648.exchange2`, which run completely out of cache and thus have unity MAR, are completely exposed to the $\frac{3}{4}\times$ slowdown brought about by the clock-token stream. Finally, in a testament to recent BOOM microarchitectural improvements, the BOOM-based experiments took less time to run than the equivalent Rocket configurations despite having average f_{emul}s of roughly half that of Rocket's.

The times we report in Table 5.3 are directly proportional to the cost of running the benchmark on AWS. At time of writing, EC2 F1 2x.large instances cost \$1.65/hr for on demand instances, and \$0.50/hr (typically) for spot instances. Considering just the BOOM designs, total emulation time was 164 and 251 hours for the synchronous and three-domain designs, respectively: on the spot these experiments would cost \$82 and \$126. One implication of AWS's cost model is that there is no disincentive to using more FPGAs to improve simulation throughput. While Table 5.3 reports an enormous runtime for `657.xz`, this is the sum of its two roughly balanced inputs, so we split the benchmark into two parallel runs to better balance the runtime of the suite. Since this is more difficult to achieve using the SPEC-provided `runcpu` utility, in these experiments we are directly invoking the benchmark binaries using Speckle-generated scripts. All told, `605.mcf` is the longest running benchmark on Rocket (28.0 h and 44.3 h) versus the CPU2006docs input of `657.xz` on BOOM (23.4 h and 35.8 h).

Target	**perlbench**		**gcc**		**mcf**		**omnetpp**		**xalancbmk**	
	time	f_{emul}	time	f_{emul}	time	f_{emul}	time	f_{emul}	time	f_{emul}
BOOM-1D	19.8	64.5	21.6	57.0	22.7	53.1	11.8	60.3	8.5	61.9
BOOM-3D	29.5	44.9	34.4	43.4	34.4	42.1	18.1	43.9	14.2	44.4
Rocket-1D	14.2	108.9	19.8	94.9	28.0	87.8	10.8	103.5	9.0	103.9
Rocket-3D	20.5	82.2	30.3	78.0	44.3	76.2	16.9	81.2	14.7	80.8

Target	**x264**		**deepsjeng**		**leela**		**exchange2**		**xz**	
	time	f_{emul}	time	f_{emul}	time	f_{emul}	time	f_{emul}	time	f_{emul}
BOOM-1D	10.0	63.5	8.5	62.7	8.9	64.4	7.1	65.0	45.3	62.3
BOOM-3D	15.0	44.4	12.5	44.2	13.4	44.8	10.3	45.0	69.3	44.4
Rocket-1D	14.7	108.4	10.7	109.0	9.5	109.5	9.7	109.9	47.8	107.2
Rocket-3D	20.4	81.8	15.9	82.3	13.3	82.3	13.0	82.5	70.7	82.2

Table 5.3: SPEC 2017 Integer Speed (single-threaded) emulation performance. We report time (wallclock) in hours, and f_{emul}, the effective emulation frequency of the cores of the design, in MHz. Note that `xz` had its inputs split over two emulators to approximately halve its runtime.

5.3 Scaling Target Clock Count

Though clock routing and distribution was enhanced in Xilinx's Ultrascale architecture, global clock resources are decidedly scarce relative to conventional logic interconnect and subject to a unique set of constraints which limit the scalability of our approach. To illustrate this we first give an overview of Ultrascale+ global clock resources and constraints imposed by Vivado, before measuring the practical limits of our current implementation in FireSim.

The Ultrascale+ clocking architecture divides a device into a regular grid of rectangular *clock regions* (CR). Each CR has vertical and horizontal tracks which can be connected to the four adjacent CRs via configurable buffers at its boundary. At the center of a CR lie switches for driving clocks from the tracks to CR-local clock sinks like CLBs and BRAMs. Global clock tracks have two flavors: *routing tracks*, which connect a clock from its source (e.g., an I/O or a PLL) to the *root* of the clock tree, and *distribution tracks*, which implement a balanced tree to deliver the clock from its root to all CRs in which there are clock sinks. To feed this interconnect, global clock buffers and clock-generation blocks, like PLLs, are vertically striped down the device in physical layer (PHY) blocks. These feed into global routing and distribution networks in adjacent CRs. In each these PHY blocks there are 24 BUFGCEs—the primitives we rely upon to implement target-clock gating.

To implement our simulators, by default Vivado tries to match delays through each clock tree carrying a target clock. It places all BUFGCEs for target clocks, as well as the global buffer (BUFG) to supply the ungated clock, in the same PHY block. To minimize clock skew between each domain, these buffers drive trees rooted at the same CR, and each tree spans the same rectangular region of CRs. Critically, this means that all target clocks, even those whose sinks reside in a single CR, consume tracks in all other CRs with sinks belonging to another target clock domain. Given our current implementation, this puts an upper bound on the scalability of our approach of 23 target clocks (alongside the emulator clock). Unfortunately, this upper bound neglects periphery IP, like DRAM controllers and PCI-E interfaces, which consume CR resources that may fall within the rectangular region required to implement the target-domain clock trees.

To establish a practical upper bound for the scalability of our current implementation, we synthesized a toy target consisting a shift register with each of its stages driven by a different clock. Thus for an n-stage shift register, n BUFGCEs will be instantiated in the hub unit. Since the number of clock sinks per domain is small, this will produce an optimistic result: the size of simulator clock trees will be defined by the location of sinks in the main emulation clock domain. Given no additional bridges are instantiated, this is close to the smallest class of design one can push through Golden Gate.

Using Vivado 2018.3 and FireSim's custom AWS shell (based off version 1.4.8), we found that we could synthesize a 12-stage pipeline before running out of global clock resources in at least one CR. In this CR only 13 of 24 available nets are used for the implementation of the hub model: the remaining 11 are used to route clocks for IP blocks and other AWS-shell-related utilities. Excluding the clocks used in Golden Gate-generated RTL, median global clock-net utilization across the device is 11 (ranging between 9 and 23) across the 90 available CRs of our VU9P FPGA. This will prove problematic for simulating systems with dozens of clocks, which is not unheard of in modern SoCs.

There are a few means to address this challenge. The first is to simply reduce the number of unique clocks required in each CR with a more optimized resource allocation. One way to achieve this is to break Vivado's default behavior of putting gated target clocks in the same clock group (so as to produced matched delays). This would permit smaller clock domains to have proportionally smaller clock trees, freeing up significant global clock resources. The downside is that this may significantly hamper Vivado's ability to meet timing constraints on paths that span clock domains. That said, our simulators tend to have lower f_{max} than many FPGA applications and may therefore be more robust against intra-domain variability in clock delay. Alternatively, it may be possible to optimize the AWS shell to reduce the number of global clock resources it requires, or better isolate its clock domains from the rest of the simulator's. Expending approximately half of all available clock routing resources on the FPGA shell represents an unfortunately large overhead. Clock utilization overhead should be an important consideration when porting FireSim to new FPGAs in the future.

A second approach is to consume fewer global clocks by using finer-grained clock gating where possible. Vivado has native support for gated-clock conversion, a historical pain-point in building FPGA prototypes, wherein it synthesizes behavioral clock-gates or specially annotated clocks into an FPGA-optimized equivalent based on the size of the clock domain. For clocks with large numbers of sinks, it appears to use BUFGCEs much like we do. For smaller numbers, it may use clock gates on leaf clocking resources (not exposed to the user), or finer still, the clock enables on logic elements and block rams themselves. This would permit Vivado to avoid the use of a BUFGCE in cases where we use one currently. In the event Vivado cannot effectively perform this gated-clock conversion, finer-grained clock gating can be implemented in Golden Gate itself by adding clock enables to stateful elements in FIRRTL, much like MIDAS's FAME transform.

5.4 Improving Simulator Performance with Multi-Cycle Setup Constraints

On the surface, one challenge with using a single host clock is that all paths in the target must close timing at the emulator frequency. In effect, Vivado will use the same setup relationship to close timing on paths whose delays are many times longer than that of the critical path in the fastest clock domain. FPGA prototypes avoid this problem entirely since each domain is driven with clocks of the same relative frequency as would exist on the SoC, giving slower clock domains more time to meet setup constraints. When this is combined with the fact that out-of-phase edges in slower domains can be launched between fast-clock edges, FPGA prototypes have a considerable performance advantage over the equivalent multi-clock FireSim emulator.

One way to close this gap is to improve f_{fpga} by giving Vivado more setup margin on paths in slower target-clock domains. Here we exploit the observation that, given our current clock-token generation scheme, clocks in all but the fastest target-clock domain cannot have positive edges launched in back-to-back host clock periods. This implies that data launched in one domain will not be captured by state elements in the same domain until at least two host cycles later. We can indicate this to Vivado using a multi-cycle setup constraint on all timing arcs that are internal to that domain. In many cases this setup relationship can be larger than two periods. A clock that has a frequency one n^{th} that of the fastest clock, can be given a setup constraint of n periods. Additionally, the presence of other clocks with frequencies that are not integer divisions of the fastest clock can further lengthen this constraint. Since the clock token stream is known at compile time, Golden Gate can analyze these relationships and emit target-specific multi-cycle constraints. In principle, similar constraints could be emitted for inter-clock-domain paths, but this would require the presence of intermediate clock-tokens between the positive edges of both domain clocks.

To illustrate the potential of these relaxations, we manually applied a two-cycle setup constraint to all intra-domain paths in the DRAM and uncore domains for the three-domain targets we evaluated in Section 5.2. We then resynthesized both sets of designs, with the constraints and without, using overconstrained host frequencies to establish an emulator f_{max}. We report the improved simulator frequencies in Table 5.4. The Rocket-based design saw greater improvements than BOOM for a few potential reasons. Firstly, in an ASIC technology, Rocket can be realized at higher frequencies than BOOM. Second, delays through common structures in OoO microarchitectures scale more poorly when mapped to FPGAs relative to in-order designs which lack many of the same structures. Taken together, the ratio

Design	**Unrelaxed** f_{max}	**Relaxed** f_{fmax}	**% Difference**
Quad-Core Rocket	112.5	168.1	49.4%
Single-Core Large Boom	70.5	90.0	28.5%

Table 5.4: Simulator f_{max} for the designs of the SPEC performance study without (*Unrelaxed*) and with (*Relaxed*) multi-cycle setup constraints applied.

Design	**Unrelaxed**			**Relaxed**			
	f_{fpga}	FMR	f_{emul}	f_{fpga}	FMR	f_{emul}	Speedup
Quad-Core Rocket	110	1.45	76.0	150	1.49	107.3	1.41×
Single-Core LargeBoom	60	1.57	38.2	90	1.65	54.5	1.43×

Table 5.5: Simulator performance for the designs of the SPEC performance study without (*Unrelaxed*) and with (*Relaxed*) multi-cycle setup constraints applied. Note, the BOOM result is arguably optimistic since the unrelaxed variant still has a fair amount of setup margin under a good initial placement.

of core path-delay to uncore path-delay is closer to parity in BOOM, than in the Rocket-based design where it is closer to the expected one-to-two ratio one would realize in the equivalent ASIC.

One consequence of increasing f_{fpga} is that various extra-emulator domain delays will now appear to have greater latency (e.g., FASED's host DRAM requests will take more cycles to be served), which could increase FMR. To provide some perspective on this effect, we booted Linux on instances of the relaxed and original designs that passed timing (we used the same simulators from the SPEC performance experiment). We report the change in simulation performance in Table 5.5. Linux boot differs from SPEC in that it has periods of sustained disk and DRAM use and so tends to have a greater FMR. While FMR does increase in the relaxed simulators it only modestly reduces the improvement brought about by the increased f_{fpga}. Were we to re-run the SPEC performance study from Section 5.2. we estimate the multi-clock designs would take roughly as long to simulate as the synchronous ones, permitting a SPEC run to complete in approximately a day.

5.5 Future Work

The support for multiple clock domains described in this chapter and released in FireSim 1.10 is an important development over previous iterations of the compiler, as it eliminates a key barrier to doing realistic pre-silicon performance validation. Within the assumptions laid out in Section 5.1.1, there are still a number of improvements we could explore.

1. **Automatic generation of target-specific, multi-cycle setup constraints.** As we showed for a limited set of designs in Section 5.4, multi-cycle setup constraints can provide an enormous performance benefit. In the future, we plan to have Golden Gate

emit target-tailored constraints during compilation so these benefits can be automatically passed on to the user.

2. **Improve best-case FMR by combining clock tokens**. Here we leverage the observation that if the state updates brought about by clock token n are not observable by any domain scheduled to fire in clock token $n+1$, it is safe to fuse these two tokens into single token with the union of their bits set. In many cases, this will address the performance cost described in Section 5.1.7. In that example, clock tokens that fire only the DRAM clock can be fused into prior tokens that fire only the tile clock, and vice versa. This would restore the best-case FMR to 1, since all tokens would carry a tile clock edge. However, this optimization does not come without cost. First, it complicates any debug utility that wants to observe target state at time between the edges scheduled in fused token. Second, it may preclude applying the multi-cycle setup constraints described above to some domains (in our example, the DRAM domain now fires in back-to-back host cycles), potentially offsetting some of the benefit of the optimization. From an implementation perspective, producing a simplified connectivity graph that could be fed into the clock bridge generator is relatively straightforward process in FIRRTL and so this optimization can be easily explored.

3. **Investigate schemes for improving clock scalability.** In Section 5.3, we proposed two means to improve the number of clocks we can simulate in a single target: place and route global clocks more efficiently, and use fewer global clocks where possible. While these should be evaluated on the Xilinx VU9P available in AWS, the portability of any approach should take into better account the feature set available in Intel FPGAs, which we have neglected in this work.

4. **Runtime-configurable target clock frequencies.** To avoid needing to synthesize multiple bitstreams to evaluate different clock frequency selections, we could make the rational relationships encoded in the clock bridge runtime programmable. Note, the number of clocks and target clock-crossing circuitry would remain fixed, which limits the utility of this feature. From an implementation perspective, this feature would require propagating clock domain information to relevant bridges at runtime instead of at compile time, but otherwise this feature is straightforward to implement.

Ultimately, some of the assumptions we outlined in Section 5.1.1, could be relaxed without a large-scale reimplementation. For instance, the clock bridge can easily be modified to generate tokens for any clock whose edge times are known *a priori*. Lifting effectively *all* of these constraints, most notably to support dynamically changing clock frequencies, is a more challenging undertaking that requires a rethinking of the clock bridge abstraction. This is the subject of the final chapters of this dissertation.

Chapter 6

Distributed Simulation of Clocking Structures via Explicit Timekeeping

The fundamental assumption made in the previous chapter was that the clocks of the target cannot change as a function of the simulated behavior of the target itself. This permits generating a clock-token stream that can run ahead of the rest of the simulator and gives good simulation performance. Lifting this assumption introduces a performance-critical feedback loop into the emulator: scheduled clocks determine when target state updates occur, but those updates affect when target clocks are scheduled.

Academic work using hardware acceleration to rapidly simulate these sorts of systems is lacking. However, one FPGA-based example for studying SoCs that can perform DVFS was built by Mantovani et al. [72]. This work is essentially a highly configurable FPGA prototype: it presupposes a particular target organization (a grid of tiles interconnected with a network-on-chip) and directly instantiates FPGA-specific clocking primitives. Using FPGA clocking primitives directly skirts the performance challenge we outlined above, allowing their system to run fast (100 MHz on a Virtex 7 FPGA). This makes it an attractive platform for research on new DVFS policies, but not generally useful for pre-silicon verification or performance validation of an SoC which does not derive from their input RTL. For examples of FPGA-based hardware emulation of systems that support dynamic frequency scaling, one must look to industry. While Mentor Veloce and Synopsys ZeBu both have support for this form of frequency scaling, their underlying implementation is proprietary.

The approach we outline in this chapter is radically different from known prior work. Our simulators have thus far represented a restricted form of parallel discrete-event simulation, wherein explicit timekeeping can be eschewed—they are ITDC. Since the time interval between tokens is fixed, timestamps are not required, saving considerable FPGA resources. In PDES terms, these simulators are conservative, as units do not speculate ahead in time, and deadlock-avoiding, as LI-BDN and channel-bootstrapped formalisms avoid deadlock by construction. However, the simplicity of these implementations derives from their assumption that the target is an SSM. So, to simulate non-SSM circuits like the structures required to support DVFS, our approach *selectively* returns to a more general “de-optimized” form of PDES.

Concretely, we achieve this by introducing *timestamped units* (TUs) which like conventional untimestamped units (UUs) are promoted and extracted into separate latency-insensitive modules. TUs communicate between themselves and the hub by sending messages over FIFO channels. Deadlock avoidance within this subgraph is maintained using null messages à la the Chandy-Misra-Bryant [18] algorithm. This approach supports a large class of non-SSM digital circuits, as it makes few assumptions about the underlying physical process modeled in a given TU.

In our implementation, we use no FPGA-specific clocking resources beyond a clock gate, making our implementation portable across different FPGAs. Finally, our approach maps well to the inherently parallel nature of clock generation and selection in a SoC. However, this implementation strategy begets its own difficulties, notably that, as in all conservative PDES, achieving good performance without deadlocking hinges on finding sufficient lookahead. In some cases this may not be possible without restricting target behavior.

6.1 Context, Goals and Motivation

To understand how we arrived at our implementation, we update the goals we outlined when we designed Golden Gate (Section 4.1) given the current capability of FIRRTL to express clocking primitives, and the machinery available in Golden Gate to implement them. These goals were:

1. **Maintain FireSim feature completeness.** Just as in the original redesign, it was critical that support for dynamic frequency scaling function in conjunction with existing bridges and multi-cycle resource optimizations. We were willing to accept a loss of simulation FMR in exchange for more rapidly obtaining a working prototype.

2. **Minimize user modification of ASIC RTL.** Specifically, we wanted to avoid extensive changes to the design's module hierarchy. The use of a singleton clock bridge in the previous chapter violated this objective. Here, we were willing to replace ASIC clock-generating circuits with inexact models, but we wanted to avoid centralizing all clock generation at a single point in the design.

3. **Provide a mechanism to rigorously verify simulation models.** While, as in the LI-BDN case, it would be ideal if parts of the simulator could be verified against the underlying ASIC implementation using a LIME-like flow, we leave this to future work. In this chapter we relied on dynamic verification to test our clock-generating units.

4. **Enable the use of FireSim as a library for hardware emulation.** This is closely related to the second point. Clocking and reset structures remain the single largest point of divergence between real chip configurations and FireSim-modeled configurations of Chipyard SoCs. Minimizing ASIC modification is critical to empowering FireSim's use as a hardware emulator.

5. **Avoid the use of FPGA-specific Clocking Primitives.** In the interest of supporting other FPGAs in the future, and to avoid more of the placement and DRC challenges associated with using those devices, we wanted a design that used conventional logic and interconnect resources where possible.

At time of writing, native support for clocking structures in FIRRTL is nascent and designers tend to use black-box Verilog to describe structures that act on clocks (these tend to have process-specific implementations that will be used in physical design anyways). Building a compiler capable of analyzing a hybrid FIRRTL-Verilog netlist without designer help was going to be challenging, so we elected to leave that to future work.

6.2 Other Designs We Considered

To reuse most of our work in building out support for simulation of systems with fixed clocks, we initially tried to frame this new challenge as solving the smaller problem of reconstituting the clock-token stream. The most straightforward way of doing so is to a provide a means for users to specify that certain clocks have dynamic behavior on the centralized clock bridge. For these clocks, the clock bridge would accept parameters to describe how it derives from other clocks in the system (e.g., it should in effect simulate a clock multiplexer and select between two other output clocks). Based on this selected behavior new inputs on the bridge would be exposed and driven by target RTL. This would require the smallest modification to existing infrastructure.

While this would suffice for simulating simple systems, the main problem with this approach is that it forces the designer to re-express, in a centralized fashion, how all clocks in a system are generated. This runs counter to the inherently distributed process of clock generation and distribution in an SoC. In a simple system, a clock source might feed a centralized PLL with closely coupled output dividers, which in turn may drive downstream dividers, clock multiplexers and ICGs distributed throughout the chip. This simplified model is complicated by I/O devices which tend to provide their own clock generation schemes that derive from still other off-chip clocks. As such, it would be better if information about target clocks could be extracted from the instantiations of ASIC clocking circuits and accompanying design constraint files, which already encode these clock relationships for other back-end implementation flows, instead of forcing the user to re-express these properties at the clock bridge.

With this in mind, we set out designing a system that would transform or replace existing ASIC circuits with FPGA-compatible equivalents. As in the previous chapter, this cannot be achieved simply by substituting a primitive with an FPGA analog, but instead requires a custom representation of that circuit that is deterministic under variable host timings.

The defining design decision of this project was whether to replace these clock-generating circuits with independent latency-insensitive units or whether the compiler should attempt to centralize them into a single chunk of logic integrated into the hub. In effect this reduces to a question of distributed versus centralized control. We chose a distributed approach for the following reasons. First, it was a natural extension of the Golden Gate framework. Replacing modules with decoupled units slots neatly into the existing abstractions and so reuses much of the existing infrastructure. Second, our abstraction is flexible enough to capture the behavior of practically all non-SSM digital circuits that FireSim cannot currently support. This uniformity made it easier to reason about correctness of pieces of the target independently, without having to "special case" support for certain types of clock primitives.

6.3 Implementation Overview

To summarize, our approach replaces non-SSM modules, notably those related to clock generation, switching, or gating, with timestamped units (TUs). Strictly speaking, both timestamped and untimestamped units (UUs) are logical processes (LP), however in this chapter we will use LP and TU interchangeably. TUs are extracted and replaced in the same fashion as UUs, like a bridge or an optimized model, however, their channels are labeled as *timestamped*. These channels carry messages between TUs, which consist of data value and a timestamp. Analogously, tokens can be thought of as implicitly timestamped messages, however we'll reserve the term messages for timestamped transmissions.

Now, simulator graphs have two halves: an untimestamped subgraph, which includes all of the optimized models and conventional bridges described in previous chapters, and a timestamped subgraph, which includes the TUs. The hub is the only unit that resides in both subgraphs and serves to reconcile the two domains. We show an example graph in Figure 6.1.

For this approach to have acceptable performance, we needed to permit TUs to send messages directly to one another, instead of forcing these transmissions to propagate through the hub. To do so, we introduced a compiler optimization that removes passthrough connectivity in the hub, permitting two units (timestamped or untimestamped) to drive each other directly. As a result, simulator graphs are no longer guaranteed to have a star topology (this is visible in Figure 6.1). While essential for the prototype described here, this optimization also improves FMR in designs described in previous chapter when multiple optimizations are enabled. We expand on this optimization and other compiler modifications in Section 6.9

Since there is no longer a clock token stream, we modified the hub unit to schedule across multiple timestamped inputs. Timestamped inputs can be either clock-type or data-type (used for modeling asynchronous resets), these differ in how they are presented to target RTL. The hub unit explicitly tracks target time, which it uses to populate the timestamps of output messages as it advances. We expand on the hub-unit modifications in Section 6.8.

TUs are implemented as bridges that use timestamped channels exclusively. From the RTL designer's perspective, they are employed no differently than conventional bridges in the target design. However, writing TUs that were both correct (deterministic, deadlock free, and timing exact) and performant we found to be the most challenging aspect of our approach. To ease this process, we wrote a Chisel library of timestamped circuit primitives

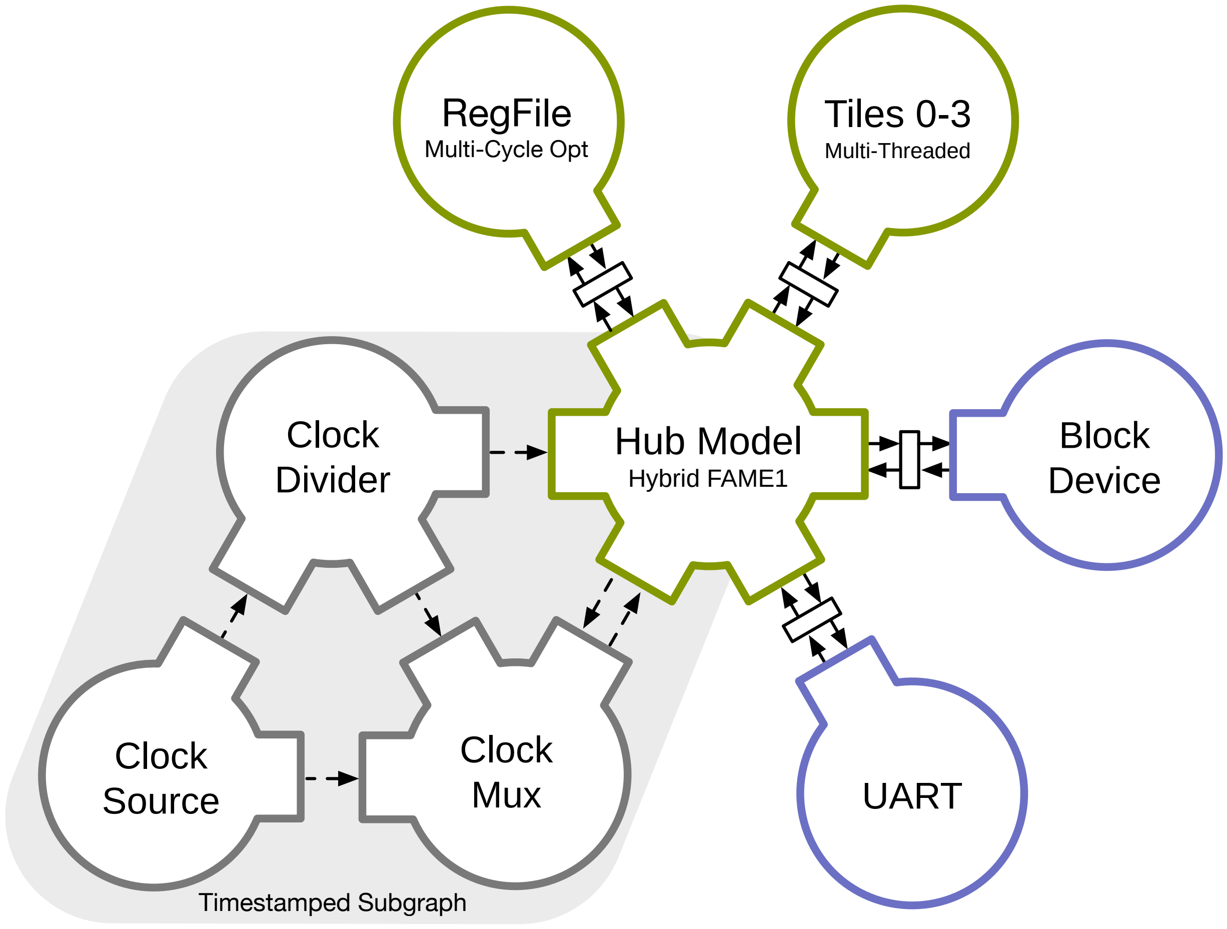


Figure 6.1: An example simulator graph with a timestamped subgraph responsible for modeling clock generation. Note, the ports represented in timestamped models are logical. The clock divider and clock mux both have a single output: output messages are duplicated and transmitted to each sink over separate channel.

to make it possible to write simple TUs as translations of Verilog RTL. Unfortunately, these primitives are conservative and cannot exploit lookahead that might emerge in their composition. We describe these baseline TU implementations in Section 6.10.

6.4 Simplifying Assumptions

While our approach imposes few restrictions on what an individual TU models, some new complexities arise when considering interactions that span timestamped and untimestamped regions of the simulator. In our initial prototype, we make the following assumptions:

- Untimestamped channels remain synchronous in that they observe the hub only on positive edges of their associated clock and once at time zero. For example, if the input to a unit is driven by asynchronously reset state the sink unit will never see a token launched by the assertion of that reset. If these transitions must be modeled with sub-cycle accuracy a TU must be used.
- All clocks must be sourced by TUs. Clocks cannot be generated in FIRRTL with type casts from other data types.
- All stateful target hardware in the hub or in UUs must be positive-edge triggered. While this restriction can be easily relaxed in the future, for the time being level-sensitive and negative-edge-triggered hardware need to be modeled in TUs or replaced with positive-edge-triggered equivalents driven by an inverted clock (if applicable).

6.5 Target-Side User Interface

To deploy a TU, the user annotates a clock-generating module with a `BridgeAnnotation`, and divides its target-side interface into channels. The bridge annotation calls out a specific `BridgeModule` class implements the desired timing model. In Listing 6.1, we show an example of this annotation process, using our library implementations of a clock multiplexer and divider as examples.

```
// The existing clock divider module instantiation
val clockDivider = Module(new rocketchip.util.ClockDivider2)
clockDivider.io.clk_in := fullRate
val halfRate = clockDivider.io.clk_out

// Annotate the divider indicating it can be replaced
// with a TU. This is a strict addition, and requires
// no other modification to the design

BridgeableClockDivider(
  // The first parameter provides the module
  clockDivider,
  // Successive parameters provide the mapping
  // from target signal to timestamped channels
  clockDivider.io.clk_in,  // Input
  clockDivider.io.clk_out, // Output
  // The BridgeModule's constructor parameter
  div = 2)

// A similar example, using a clock mux
val clockMux = Module(new testchipip.ClockMux2)
clockMux.io.clocksIn(0) := fullRate
clockMux.io.clocksIn(1) := halfRate

// Annotate the clock mux, as above
BridgeableClockMux(
  clockMux,
  clockMux.io.clocksIn(0),
  clockMux.io.clocksIn(1),
  clockMux.io.clockOut,
  clockMux.io.sel)
```

Listing 6.1: An example of the target-side modifications required to call out clock-generating primitives as units.

6.6 Message Representation and Channel Design

In our system, a signal that spans two TUs is represented with a trace of messages, each labeling a transition with the time at which it occurs. In hardware engineering terms, this in effect defines a two-state value-change dump (VCD) for the signal. While there are many ways to optimize message encoding, logically all messages in our system have two fields: a timestamp, which is a 64-bit unsigned integer representing an absolute time in a simulator-global timebase, and data, which represents the value of the signal at the associated time.

For a concrete example, consider a clock. A clock-type message consists of the 64-bit timestamp and a boolean data field. To represent the clock over the duration of a simulation, a source must send a message on every transition, so to encode N periods the source must send at least $2N$ messages. We say "at least", because we rely on the Chandy-Misra-Bryant [18] deadlock-avoidance algorithm, and so practically speaking, many message streams will include null messages. In our implementation, a null message shares the same data value as the most recently sent message but with a later timestamp.

In our initial implementation, all timestamped channels are implemented with a single fully decoupled queue. This means they can transmit at most (i.e., enqueue or dequeue) a single message per host cycle. Without further optimization, this implies that simulator FMR has a lower bound of two, since any unit processing a clock-message stream must handle a negative-edge transition in every other cycle. While there are many approaches that can overcome this limitation, which we discuss in Chapter 8, in this chapter we work within this simple but general representation.

6.7 Correctness Of Logical Processes

In their paper, Chandy and Misra [23] provide a formal specification of their null-message-based conservative PDES and define its correctness vis-a-vis the physical system it simulates. We provide an informal summary here. As we introduced in Chapter 3.5, a physical system consists of a graph of communicating physical processes. In our systems, physical processes are digital circuits of any scale, that communicate by driving wires that connect them.

To reason about the correctness of TUs, much like LI-BDNs, we must frame them against output traces generated by their references (PPs). Chandy and Misra dub the sequence of all messages sent from source i (PP_i) to sink j (PP_j) up until time t, as a message *history*, $h_{ij}(t)$. LI-BDN's have an analogous definition of history: their "messages" simply lack explicit timestamps, and t is specified in cycles. In the logical system, the message history between the equivalent source and sink LP is given by $H_{ij}(t)$, where analogously, $H_{ij}(t)$ is the sequence of messages sent from LP_i to LP_j up until time t. Crucially, H and h differ in that h can never contain null messages: these are an artifice of the simulation required to avoid deadlock.

Chandy and Misra say a message history between a source and sink LPs is *correct* if and only if all messages of h_{ij} are present in H_{ij}, and every message in H_{ij} either exists in h_{ij} or is a null message. In simpler terms, rejecting all null messages in H_{ij} should produce h_{ij}. This gives a concrete means to verify a TU against a piece of reference RTL: pass identical inputs (i.e., H for this input is "correct" in the sense above) to the reference module and the TU and compare the reference RTL's output transitions and their times against the output message history of the unit after removing all null messages. With null-messages removed, this notion of correctness is analogous to establishing *sequential equivalence* [83] between the reference RTL and that behavior represented by the TU.

6.8 Hub Unit Modifications

At a high-level, the primary change to the hub-unit is the introduction of a frontend to the pipeline that reconstitutes the clock token from a set of timestamped inputs. Timestamped inputs are processed in two groups, based on the FIRRTL type of their target signals:

- **Clocks:** These drive FIRRTL `clock`-typed inputs. As mentioned in Section 6.6, these are timestamped booleans. On a positive-edge transition, a dedicated clock buffer responsible for driving the target clock is enabled.
- **Data:** These drive any FIRRTL non-clock ground-type including `UInt`, `SInt`, and, notably, `AsyncReset`. Updated values are presented to the target RTL by latching new values into a register at the times indicated in the message stream. In this way, an asynchronous reset transition can be presented at a time when no clocks are active, and timestamped outputs can observe newly launched transitions in distinct messages.

We show the updated hub unit in Figure 6.2. The frontend of the control pipeline (bottom) consists of a series of decoders that unpack messages to provide the horizon over which an input is defined (`definedUntil`) and detect transitions. If an input message produces a transition it is not dequeued: the decoder sets `definedUntil` to that message's timestamp and waits. Conversely, null messages and negative-edge transitions in clock-type channels are dequeued upon arrival, which allows that input's horizon to advance further. When no message is available, the decoder holds `definedUtil` at the timestamp of the last message.

With the inputs unpacked, the frontend performs a minimum-reduction across the horizons of all timestamped inputs and an upper bound provided through the control interface (`ctrl.timeHorizon`). This becomes the candidate time for the next timestep and is broadcast back to each decoder (`advanceToTime`). If the pipeline can advance, all decoders with a transition or positive clock edge at the scheduled time dequeue their input messages, and have their updated signal value latched. The concatenation of all enabled positive edges (`posedgeEnable`) corresponds to a clock token in the sense defined in the previous chapter. Note that it is possible for there to exist no transitions in a given timestep, it is scheduled anyway for reasons we will discuss momentarily.

The "clock token" and data updates are scheduled and flow through the pipeline as in the previous chapter (note, the first stage of the pipeline for data values is absorbed into the scheduler). Untimestamped input (UI) and output (UO) channels handling has not changed, their control FSMs are only reset when their clock has been scheduled to fire. One critical implication of this is that asynchronous events launched by data-type transitions in a particular domain do not launch new tokens in an untimestamped channel. This is consistent

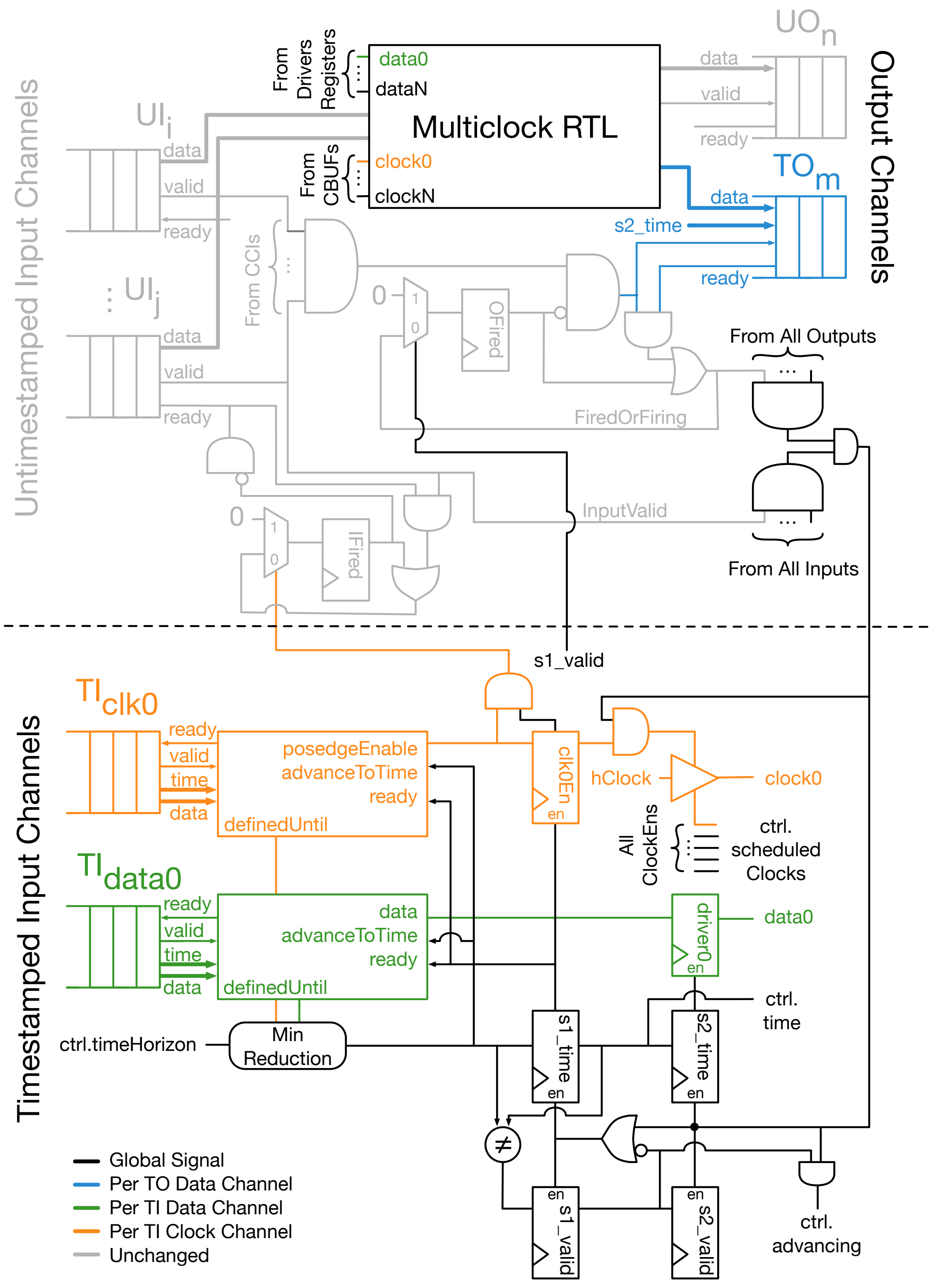


Figure 6.2: The updated hub-unit wrapper. Circuitry in grey is unchanged from the multi-clock wrapper shown previously in Figure 5.1

.

with our previous assumption from Section 6.4, that untimestamped channels only observe outputs synchronously with respect to their associated clock.

Timestamped outputs are managed differently. To avoid certain deadlock conditions, TOs generate a new message on every timestep, notably those in which no clock or data transitions have been scheduled. In effect, these outputs have their FSMs reset on every timestep (via `s1_valid`), and source their timestamp from the pipeline (`s2_time`). Without further optimization, this baseline hub unit guarantees that output channels will always be defined only as far as it oldest timestamped input. Critically, this means the hub unit has zero lookahead. Thus to avoid deadlock, any arc that spans one or more TUs and begins and ends at the hub must provide non-zero lookahead.

To provide finer-grained control and instrumentation, the hub exposes specific signals to a memory-mapped simulation controller. This controller can detect whether clocks are scheduled to fire (`ctrl.clocksActive`), the time of the associated timestep (`ctrl.time`), and is responsible for setting the time upper-bound (`ctrl.timeHorizon`) past which the hub must not advance.

6.9 Compiler and Annotation Modifications

At first blush, baseline support for timestamped channels is relatively straightforward to implement. Since all timestamped channels are wire-type, the timestamp of a message can be treated as target data in an otherwise conventional token, and wiring between channels and bridges can be left unchanged. However, to provide a workable baseline implementation, we needed TUs to be directly connected to one another, and not indirectly through the hub. This permits TUs to advance ahead of the hub unit (which in its current form has zero lookahead). Our earlier assumption that only TUs can source clocks and signals that launch asynchronous events lets us ensure that sources are only ever directly connected to their sinks (i.e., through a series of FIRRTL `Connect` statements). This in turn simplifies the process of finding and extracting these paths into point-to-point channels between two units. We refer to this enhancement as the *passthrough connections optimization* and give a pictorial representation of the process in Figure 6.3.

The core of this change revolves around the `PromotePassthroughConnections` transform, which extracts wires of any FIRRTL type that passthrough any top-level SSM. We run this pass in target transformation, after bridge and model extraction. Note, if a signal drives no other sinks aside from output ports, we leave the connection to the original unit in place instead of removing the connectivity altogether. This simplifies connectivity analyses that occur later in the compiler, and ensures that all clocks still drive the hub (even if all of the state for a particular domain has been extracted).

At this point top-level connectivity in the design now includes fanouts. To handle this new class of connectivity we modified `ChannelExcision`. In general, a fanout of n produces one output interface, and n input interfaces (sunk by the hub unit and optimized models)

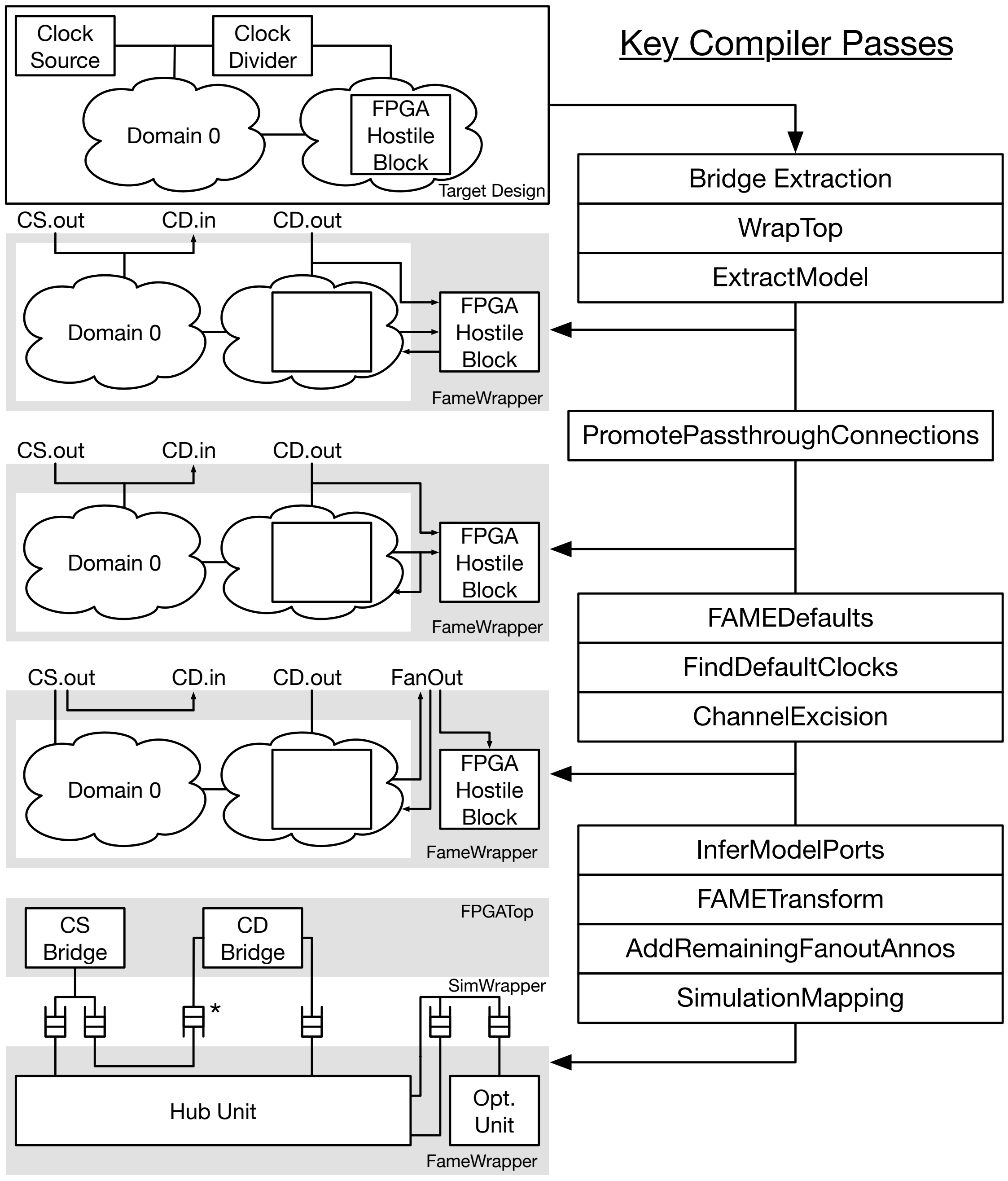


Figure 6.3: A visualization of the promote passthrough optimization. Modifications to the target are shown on the left at key points during compilation. We include a passthrough path in a optimizable block (right) to show how this transformation is broadly applicable to all inter-unit connectivity. Note, there is an extra channel generated (asterisk) in the bridge-to-bridge case currently, but this can be optimized away in a future change.

on the FAMEWrapper (see the right-most fanout in Figure 6.3). The resulting channel annotations share the same `source` targets, but their `sink` targets point at unique inputs.

While a model-driven fanout can be deduced by looking for matching source fields in channel annotations, this does not apply to a bridge-driven fanout, since bridge-driven channels leave their `sources` field empty. Since bridges are already removed (they are interfaces on the FAMEWrapper), a bridge-sourced fanout of n produces n inputs and no output interfaces. The clock-source-driven interface (CS.out) in Figure 6.3 is an example of this. To re-associate these inputs with a single source, we collect bridge-driven sources in fanout annotations (`FAMEChannelFanoutAnnotation`). This allows simulation mapping to drive the correct set of input interfaces from the same bridge source. For consistency, we emit the same annotations for unit-driven fanouts (`AddRemainingFanoutAnnos`).

Since this optimization now removes passthrough clock paths in extracted SSMs, an SSMs's target clock will be driven not from the hub, but instead from a TU-driven input (the path from CD.out to the FPGA hostile block in Figure 6.3). This required a modification to `FindDefaultClocks` to infer the clock fields of intra-UU channels by finding a clock sink port on the hub that shares the same source as the clock sink on the extracted SSM. This works because, as we previously suggested, `PromotePassthroughConnections` ensures that all clocks remain driven to the hub.

The final necessary change for this optimization was to generalize the `FAMEtransform`'s re-wiring of the FAMEWrapper, which previously relied on the assumption that all channels had to be sourced by a unit and sunk by a top-level interface (or vice versa). Now inter-unit connectivity, like the connection between the CS and CD in Figure 6.3, must also be rewired.

The optimization, while critical for the implementation of this dissertation work, can improve FMR in simulators without a timestamped subgraph notably when using multi-cycle optimizations inter-model or bridge-to-model, paths need not propagate through the hub model first. In a target in which a passthrough path would winds through the hub, through an extracted model, and back to hub (with no other combinational paths that span the same arc), FMR improves from a best-case of three to two. This improvement becomes more pronounced for paths that run through a cascade of connected models. This was the first major contribution of the work of this chapter back to mainline FireSim (the optimization is available in version 1.12).

6.10 Baseline Timestamped Units

Beyond a doubt, writing TUs was the most difficult aspect of building out our prototype. Many of the aforementioned difficulties associated with writing UUs apply. TUs must not only be robust against changes in message arrival times (latency insensitive), but they must be able to tolerate the presence of additional null messages. In UUs, the designer must manage cycle-scale decoupling (for example, that the output has advanced some number of cycles ahead of the input), whereas in TUs this decoupling is considerably finer-grained. Additionally, whereas SSMs without combinational loops have well-defined outputs for a given input trace, this is not true of many of the circuits we wish to model here. One typical example of this is that there can often be race conditions in standard Verilog implementations of these circuits that arise due to the simultaneous arrival of inputs. This can produce non-deterministic simulation behavior that is still compliant with Verilog's event-ordering model. Since determinism is critical for making FireSim simulations debuggable, the TU designer is forced to pick one behavior. Finally, TUs must extract sufficient lookahead from their reference RTL to avoid deadlock and provide good simulation performance.

We built our baseline TUs using a modular approach that would make it easy to rewrite Verilog implementations as TUs. Specifically, we built a library of timestamped utilities to manage and unpack message streams, circuit primitives for registers, single-output combinational logic functions, and fanouts. For each of these primitives we wrote deterministic Verilog reference implementations, and built timestamping hardware capable of translating the reference output into a message stream. This let us compare TU outputs with reference-generated message traces as part of a dynamic verification flow.

A key limitation of these initial implementations is that they omit asynchronous reset. While not insurmountable, asynchronous reset poses a key challenge to our approach as it tends to remove lookahead opportunities afforded by target state. In lieu of this, we rely on FPGA programming or host reset to put TUs in a desired initial state. We discuss the complexity of modeling asynchronous reset in these TUs in Future Work (Chapter 8).

6.10.1 Timestamped Tuples: Unpacked Messages

Message streams are difficult to directly act upon, as in general, we are often interested in data values between messages. One reasonable approach is to insert incoming messages into an age-ordered event queue (as a software implementation would) and update TU state in message order. In our TUs we took a distributed approach in which we wrote ad-hoc state machines to schedule over per-input datastructures to avoid serialization on a single event queue. Given our baseline message encoding it is not possible to detect transitions in a signal without comparing it to an older value, so these hardware datastructures, which we refer to as *timestamped tuples*, ease this process by holding a pair of messages: the latest message (the head of the channel's queue), and the previous message. When a unit wishes to process events created by a non-null input message, it asserts `observed`, which moves the latest message into the secondary slot and the next message in the queue becomes visible. At that point the former `previous` message is no longer visible and cannot affect output messages and internal state changes.

Unlike early conservative PDES work, in which lookahead in an LP is statically defined, our TUs can dynamically extract lookahead based on the values of currently visible inputs. This is a domain-specific optimization akin to that first described by D. Nicol [75] and B. Lubachevsky [69]. To describe lookahead properties of our library primitives, it is useful to define variables that derive simply timestamped tuples. Suppose we have an input I, we define:

- T_I to be the time of the latest message. We refer to this as the *horizon* of I.
- P_I to be the time of the previous message.
- I_H to be the data value of I at its horizon.
- I_P to be the data value of I before its horizon, and defined at least as far as P_I.
- $I(t)$ to be the data value of I at time t, where:

$$I(t) = \begin{cases} I_H & \text{if } t = T_I \\ I_P & \text{if } T_I > t \geq P_I \\ undefined & \text{otherwise} \end{cases}$$

Generally speaking, lookahead is defined with respect to the oldest input received by an LP. In many of our TUs, lookahead will be zero, notably when a clock lags lags all other inputs. However, this may be permissible if the clock input is not part of a cycle in the timestamped subgraph, or if at least one other TU in a cycle containing this input has non-zero lookahead. As we will see, of particular concern for deadlock avoidance are hub-driven control inputs to TUs. Here, we define *relative lookahead*, which is the available lookahead in a TU, when a specific input lags all other inputs. For example, select-relative lookahead on a clock multiplexer refers to the lookahead available when the select input to the mux lags all clock inputs.

6.10.2 Edge-Triggered D Flip-Flops

D flip-flops, both positive and negative-edge triggered, are critical elements for building integer clock dividers and clock multiplexers. Our basic timestamped model has three compile-time parameters: the Chisel type the register stores, the initialization value register assumes at time 0, and flip-flop's edge sensitivity. Structurally, it has a clock-type input, and a data-type input (D) and output (Q) of the data type above. This primitive models no clock-to-Q delay, and so an output transition on Q occurs simultaneously with a latching edge. Additionally, this register observes the value of D one timescale unit before the positive edge. Thus, if a clock and data transition occur simultaneously, the clock always observes the old value of D. This is often a data-race in a Verilog expression of a flip-flop as the assignment to D and the evaluation of RHS of a non-blocking assignment to Q can be executed in either order. Our implementation ensures the non-blocking RHS evaluation occurs first. We give the rules governing the timestamp of an output message based on the available inputs, and thus the available lookahead (T_L) below:

```
if $T_{clock} \geq T_D$ then
    if Clock = latching edge and $(T_{clock} - T_D) > 1$ then
        $t_L = (T_{clock} - 1) - T_D$
    else
        $t_L = T_{clock} - T_D$
    end if
else
    if $D_H = Q_H$ then
        $t_L = T_D - T_{clock}$
    else
        $t_L = 0$
    end if
end if
```

Using the information encoded in the message streams alone, this register's output cannot always advance ahead of both Clock and D. However, it does have non-zero D-relative lookahead, which will suffice to avoid deadlock in many important classes of designs we will discuss in Section 6.12. We note, the most natural way to introduce clock-relative lookahead, if required, is to model a clock-to-Q delay. Similarly, D-relative lookahead could be improved by having the model observe D earlier relative to the clock edge, as the path will need to meet a reasonable setup-time constraint to avoid metastability.

6.10.3 Generic Combinational Functions

Our baseline utility to describe combinational functions schedules over a set of timestamped tuples, and generates a single output message at the timestamp of the oldest input message. This is convenient in that the user can generate arbitrary combinational logic over all input tuples, however it comes at the cost of zero lookahead. Without providing additional hints about input transitions, there are two mechanisms for finding more lookahead. The first is modeling propagation delays through gates. Here,

$$t_L = \min_{\forall I}(T_I + minPD_I)$$

where $minPD_I$ is the minimum propagation delay of the circuit from input I to the output. The second means is to leverage logic redundancy. If the oldest input to a combinational function cannot affect the output (i.e., it can be treated as a don't-care) the output can advance ahead of that input.

6.10.4 Fan Outs

To broadcast a timestamped tuple from one source to many sinks, we wrote a fanout primitive. Under-the-hood, this repacks the tuple into a message that is then broadcast to a set of

```
module ClockDivider2 (output reg clk_out, input clk_in);

   initial clk_out = 1'b0;
   always @(posedge clk_in) begin
      clk_out = ~clk_out; // Must use =, NOT <=
   end

endmodule // ClockDivider2
```

Listing 6.2: A reference phase-aligned clock-divider from Rocket Chip used in RTL simulation.

queues, much in the same way a fanout channel is implemented in the simulation wrapper. These queues are of finite depth, and must be sized conservatively by the designer to avoid deadlock.

6.10.5 Clock and Asynchronous-Reset Sources

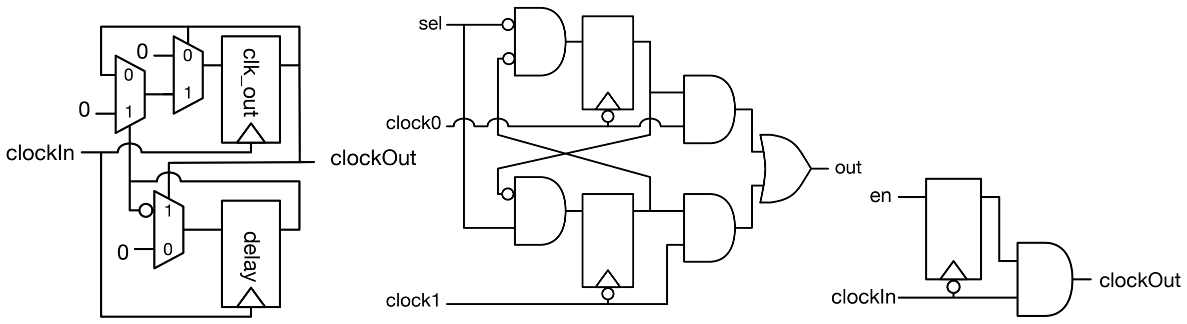
For use in test harnesses, we wrote primitives to act as idealized sources for clocks and asynchronous resets. Our clock source generates an infinite trace of clock messages, and has a compile-time configurable period, initial value, and duty cycle. Having only a single clock output and no inputs, it effectively has infinite lookahead and can run arbitrarily far ahead of other TUs in the graph. Similarly, our asynchronous reset source drives a pulse of configurable length and polarity, after which it remains deasserted-asserted for the remainder of simulation (a total of three messages). Note that asynchronous resets can also be generated by other TUs (like clocks) or by target logic itself: this primitive is useful for supplying off-chip resets that cannot be defined in terms of other signals. Both of these primitives have a wrapper bridge module, and so can be used in target RTL.

6.10.6 Library Timestamped Units

Using these library primitives we built simple TUs by translating Verilog into timestamped equivalents. We give an example of this using a phase-aligned, by-two clock divider: its reference Verilog is shown in Listing 6.2 and its timestamped translation is shown in Listing 6.3. Using this scheme, we also wrote baseline TUs for a by-three clock divider (Figure 6.4a), a glitchless two-to-one clock multiplexer (Figure 6.4b) and an AND-based, ICG (Figure 6.4c). Note that the output clock in each of these circuits is not combinationally dependent on a control input: they all have non-zero control-relative lookahead.

```
class ClockDivider2 extends MultiIOModule {
  val clk_in    = IO(Flipped(new TimestampedTuple(Bool())))
  val clk_out   = IO(new TimestampedTuple(Bool()))
  val reg = Module(new TimestampedRegister(Bool(), Posedge,
     init = Some(false.B)))
  reg.simClock <> clk_in
  val Seq(feedback, out) = FanOut(reg.q, "feedback", "out")
  clk_out <> out
  reg.d <> (new CombLogic(Bool(), feedback){
    out.latest.bits.data := ~valueOf(feedback)
  }).out
```

Listing 6.3: The equivalent TU for the clock divider shown in Listing 6.2 implemented using library primitives.



(a) By-3 clock divider. (b) Glitchless clock mux. (c) Clock-gating cell.

Figure 6.4: Reference circuit diagrams for the baseline TUs. Verilog for the by-2 divider is provided in Listing 6.2 so we omit it here.

6.11 Verification

We relied on dynamic verification to check our implementations. We wrote a SystemVerilog "timestamper" which translates a signal into a decoupled message stream, by leverage Verilog's `$time` system function to produce a timestamp. We then wrote Chisel testbenches to show that two message streams were correct in the manner described in Section 6.7.

As we have mentioned, many of the aforementioned circuits have non-deterministic behavior when written in a conventional Verilog style. To ease verification, we wrote race-free reference implementations which guarantee a particular ordering by judiciously adding sub-`time_unit` delay to statements in the reference. This approach does not scale to compositions of these circuits, but suffices to verify small circuits are deterministically represented by its TU.

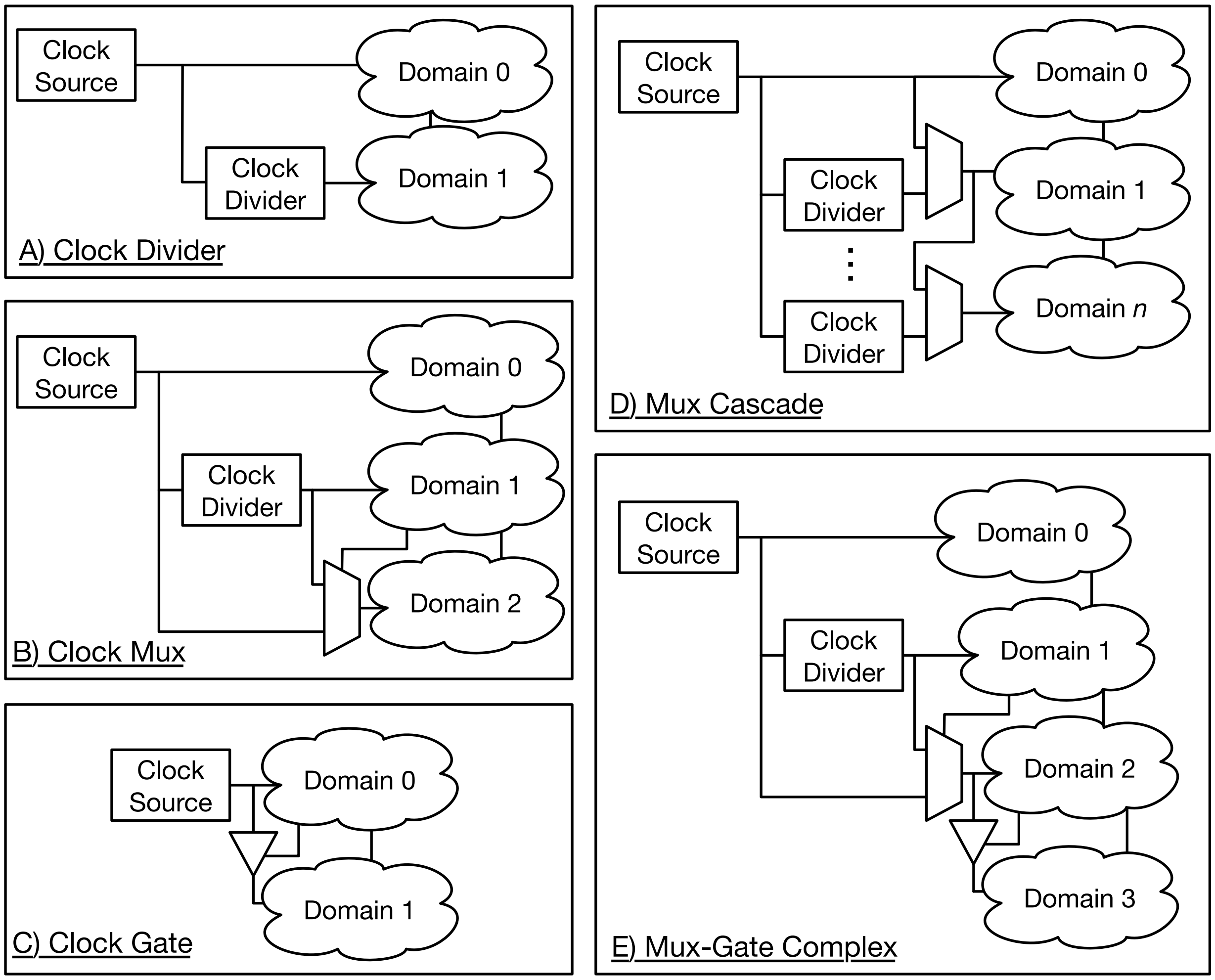


Figure 6.5: Five illustrative clock organizations that leverage each of the TUs we have built.

6.12 Performance in Common Clock Organizations

What should be clear from Section 6.10 is that the baseline TUs are pessimistic. Since they make no assumptions about the behavior of their inputs, they cannot find lookahead in all of their input corners, specifically in cases where a clock-type input is the oldest input to the unit. Fortunately, in many useful classes of clock organizations (depicted in Figure 6.5), there are no cycles that do not span the hub unit. Since all of our TUs have non-zero relative lookahead with respect to their hub-driven inputs—which are necessarily data-type—all of these circuits narrowly avoid deadlock. Of course, this does not imply that the resulting simulator is performant: here we measure the FMR of our prototype simulating the circuit organizations of Figure 6.5, and shed light on the origin of performance losses where they exist. We summarize the FMRs of these circuits in Table 6.1.

Divider	**Mux**	**Gate**	**3-Mux Cascade**	**Mux-Gate**
3.0	11.0	6.0	(12.5, 13.0)	15.5

Table 6.1: Measured FMR of the clock organizations shown in Figure 6.5 using the baseline TUs. For the 3-mux cascade, we report the FMR when domain $i-1$ drives domain i's clock select (12.5), and when domain i drives its own select (13.0).

6.12.1 Feedforward Divider Networks

A feedforward network of clock sources and dividers (Figure 6.5, circuit A) is the simplest case, and suffices to model the systems we described in Chapter 5. Performance in these circuits is bandwidth bound on the ability of the downstream dividers to process input messages from the high-frequency reference clock. The baseline by-two clock divider, reported in Table 6.1, takes three cycles on average to process two input messages (one reference clock cycle). This arises because the flip-flop model, unaware that it is driving itself in a feedback loop, waits for its D input to be defined one time unit before the next positive clock edge, introducing a single internal null message. Thus, when the fastest clock is driving other logic in the target, the best case FMR of these targets is three.

It is important to note that if we remove the fast clock domain (domain zero in Figure 6.5), from the user's perspective the FMR doubles to six (when using by-two clock divider). Unfortunately, cases in which the high-frequency clock is not widely deployed in the target-design is a common case. For example, in order to generate the 1.5, 1.0, and 0.75 GHz clocks for the three-domain target SoC we described in Section 5.2, we would be forced to generate a 3.0 GHz reference, and then use by-two, by-three, and by-four (two by-twos) clock divisions to generate the target clocks. The cost of directly simulating the fast clock increases as the ratio between the reference and the fastest derived clock grows. We describe solutions to this problem in Chapter 8.

6.12.2 Simple Hub Cycles

ICGs and clock multiplexers (Figure 6.5, circuits B and C) create a two-unit cycle in the simulator graph that span a hub-driven control signal and a TU-driven clock. Performance in these configurations is latency bound on message transmission through this cycle and is further deteriorated by null message transmissions that may require multiple serialized trips through the cycle.

This is most clearly illustrated with the clock gate configuration, which has a measured FMR of six. Both the reference clock and gated clock have the same frequency, so there no new bandwidth bound introduced. Any additional slowdown is the result of the ICG TU waiting on message propagation from the hub. We show this process in Figure 6.6. In this example the reference clock has a period of four, and there is a positive edge at $t = 0$. Since the hub must wait for input message from both the source and the ICG TU, and since the ICG TU always lags the source, we neglect the source in this diagram.

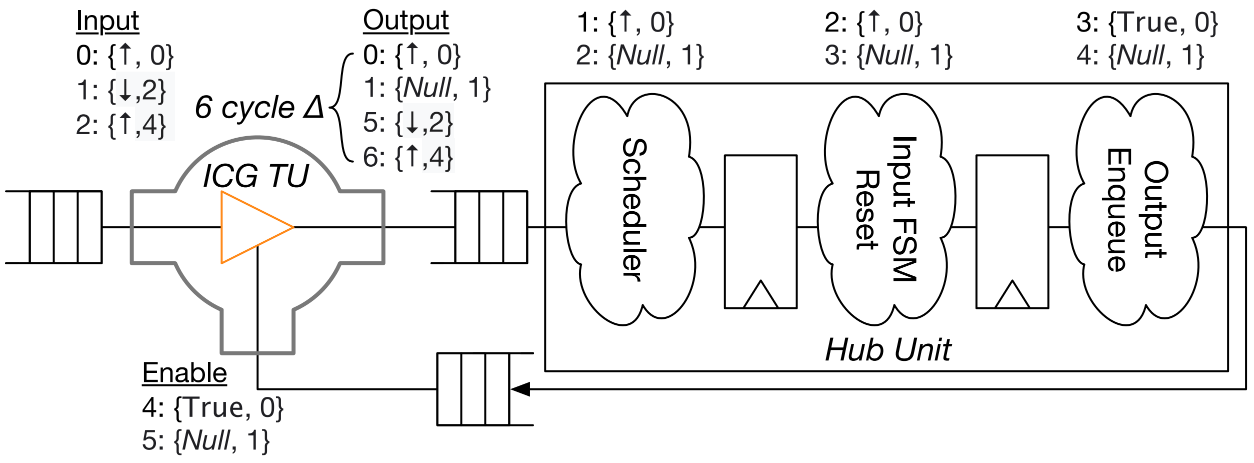


Figure 6.6: An illustration of the clock gate configuration's measured FMR of 6.0. Messages are labeled *host cycle*: $\{data, time\}$.

There are two effects at play here. After the ICG has enqueued a positive-edged message for its output clock at host cycle zero, it takes four cycles for an enable message sharing that timestamp ($t = 0$) to arrive at the TU's input. A transition in enable would be visible in this message if it was to occur. Unaware that it should expect no further transitions, the register model in the ICG waits for an input defined one time_unit before clock's negative edge ($t \geq 1$). This requires a null message (the TU's second output message) to propagate through the loop. Once it arrives at cycle five, the negative edge is processed and the new enable value is latched. In host cycle six, the second positive edge is finally enqueued. This cycle recurs thus giving an FMR of six.

The clock multiplexer configuration has significantly worse FMR (11.0) because null message propagation loops are not always overlapped with the propagation of an edge (as in the previous example). In other words, a null message received on the select input by the multiplexer launches a new null message that must propagate through the simulator, instead of allowing the unit to enqueue a new output edge.

6.12.3 Multi-Unit Cycles

Multiplexer cascades (Figure 6.5, circuit D) and multiplexer-gate complexes (Figure 6.5, circuit E) are also common occurrences in SoCs. Here a cycle starts at a hub-driven control signal, passes through multiple cascaded TUs, and terminates at a clock input on the hub. One might expect that these topologies would have greater FMRs, since the transmission time across the loop is longer without extracting substantially more lookahead per TU crossed.

This is not always case with multiplexer cascades when selecting between clocks of the same frequency: it depends on which domain drives the select signal. When the select is driven by the same domain the multiplexer drives, these circuits run with an FMR of

11.0 (note shown) independent of the depth of the cascade (N). When select is driven by the previous clock domain, the FMR increases to 13.0. When each stage is fed by a different frequency, for instance with clock divider with a division of $i + 1$, where i is the index of the current domain, the story becomes more complicated. Here increasing the cascade depth, marginally increases FMR (12.5, 13.2, 13.2, from $N = 3$ to $N = 6$), due to the introduction of new null messages by the dividers themselves. Removing these messages would return FMR to 11.0.

Topologically, the mux-gate complex is analogous to the previous-domain-drives-select mentioned in multiplexer-cascade example, as the clock enable cannot solely be driven by state in the domain it is gating. Here we measured an increase in FMR from 11 to 15.5 over the single multiplexer case, because the launch of an enable message is serialized directly behind the selection of a clock. Indeed, the primarily driver of increased FMR is not the length of a cycle, which can be problematic, but rather tight coupling clock generation and the control logic.

6.12.4 Compositions Of Circuits

We speculate that, in general, the circuits we present in Figure 6.5 can compose with zero or modest increase FMR if they derive from the same clock source, or from a second clock source that is a phase-aligned integer division of the source. This is because their message propagation happens in parallel, and few, if any, new simulator timesteps are introduced. More complex compositions that derive from a clock driven by a multiplexer require more detailed analysis, as which domains drive control signals can have a large bearing on FMR. This effect is manifest in both the mux-gate complex and multiplexer cascade circuits.

6.13 Lookahead Optimized Units

While there are systemic approaches for improving simulation performance in our initial implementation, the lowest hanging fruit revolve around optimizations that can extract more lookahead in non-hub TUs. Without introducing logic or propagation delay, this must rely upon leveraging the presence of state in the reference circuits. This state provides a multi-period interval of opacity [69] wherein changes in the hub-driven control signal are not observable in the output message stream—an effect we first described in the context of channel-bootstrapped formalisms. To this end, here we throw out the library primitives in favor of fully handwritten TUs. The cost of this approach is that writing correct TUs becomes more challenging, as these units cannot be simply transcribed from Verilog.

6.13.1 Integer Clock Dividers

If we continue to neglect asynchronous reset, it is trivial to write a clock divider TU that can consume an input message per cycle. On positive input clock edges, the TU increments

	Divider	**Mux**	**Gate**	**3-Mux Cascade**	**Mux-Gate**
Baseline	3.0	11.0	6.0	(12.5, 13.0)	15.5
Optimized Dividers	**2.0**	11.0	—	(*13**, 13)	15.5
Optimized ICGs	—	—	**4.0**	—	**14.0**
Sync Mux, $k = 1$	—	**2.6**	—	(**3.0**, **4.8**)	**5.3**
Sync Mux, $k \geq 2$	—	**2.0**	—	(3.0, 4.8)	**5.0**
All	2.0	2.0	4.0	(**2.0** **2.8**)	**2.7**

Table 6.2: Measured FMR of the clock organizations in Figure 6.5 when introducing lookahead-optimized TUs. Bolded figures represent an improvement over the previous best. While previous rows substitute only a single type of TU to isolate FMR changes, the *All* row uses optimized units exclusively.

a counter, and if the counter has reached a threshold for a positive or negative output edge, it supplies the appropriate output message. Otherwise, the TU greedily enqueues output null messages stamped to the arrival time of the last input clock edge. Assuming an input message stream has no null messages, the only circumstance under which this unit cannot process an input edge per host-cycle is when there is output backpressure.

6.13.2 Integrated Clock-Gating Cells

A typical latch-based ICG is difficult to optimize because the latch is transparent in the half-period directly before an input positive edge. As a result, late-arriving enable signals produce observable changes in the output. Indeed, the latch exists to prevent glitches in the output clock, so from a lookahead perspective it is more illustrative to think of this circuit as a combinational-AND of its inputs. To avoid this, in our primitive model, we used a negative-edge-triggered D flip-flop instead of a latch.

Without modeling new delays in the TU, the only recourse the unit designer has is to exploit knowledge about the behavior of the input message streams. Here the main observation is that the clock-enable function tends to be synchronous to the input clock. If this is the case, and the enable signal is being driven by the hub, there should eventually be an input message with the same timestamp as the input positive edge that launched it. Whereas the baseline implementation must be sure the enable remains stable until the time of the latching edge, in this case, the TU can lookahead one input cycle. This removes the need to wait for null messages to arrive that ensure there are no further changes in the enable. Referring to the FMR illustration for the clock gate organization (Figure 6.6), this optimization permits the ICG TU to enqueue the second positive edge on the same cycle the enable message for the previous edge arrives (note, the negative edge message for the previous cycle would be enqueued on host cycle one).

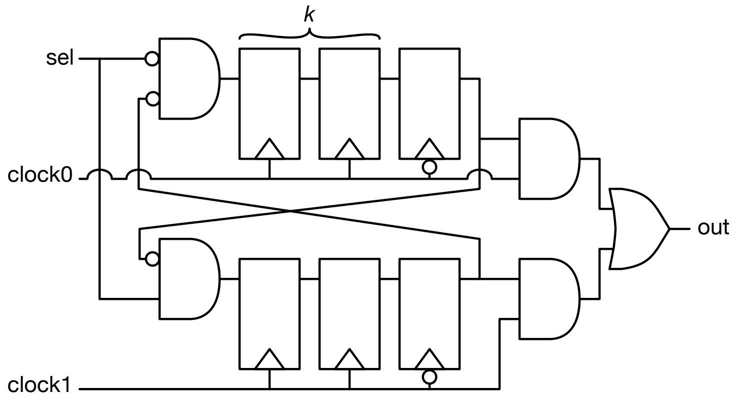


Figure 6.7: A synchronized, glitchless two-to-one clock multiplexer that can safely switch between two unrelated clocks. k is the length of the synchronizer chain.

6.13.3 Clock Multiplexers

Our baseline clock multiplexer is based on the circuit in Figure 6.4b, and extracts the smallest degree of lookahead available to avoid deadlock. While a handwritten model could do better by exploiting knowledge of the feedback loop between the two clock-select registers, to do better still, we can model multiplexers that produce observable changes in the output clock many cycles after the select has changed (they have a longer opaque period). This is true of typical glitchless clock multiplexers designed to switch unrelated clocks (Figure 6.7), which we shall refer to as a sync-multiplexer or sync-mux for short. Here the clock select must be first synchronized to each clock domain, before it produces an observable change in the output clock.

We designed a TU unit to model this variety of clock multiplexer. Our implementation is configurable in the number of input clocks (n), and the depth of the synchronizer chain (k). Internally, this TU consists of n clock handlers which dequeue input messages up to k positive edges in advance of the select. In steady state, a clock edge k periods in the future of the current clock select value can be driven to the output. On a transition no further clock edges are presented. Instead, after a negative edge for a currently selected clock has been sent, the TU sends a null message corresponding to a dead window of k periods in the new clock domain. At this point, positive edges for the newly selected domain are made visible as they arrive at the clock handler. The choice of k is the key determiner of the available lookahead in this variety of multiplexer. In Table 6.2 we present FMRs for both $k = 1$ and $k \geq 2$ cases. While the $k = 1$ case already provides an enormous improvement over the baseline, it is typical to set $k = 2$ or $k = 3$. Under these configurations, the FMR cost of the multiplexer is almost entirely removed.

TU	Logic LUTs	Registers	Memory LUTs	f_{max}
Clock Source	38	63	0	200
ICG Baseline	1134	283	84	128
ICG Optimized	260	131	0	200
Clock Mux	3372	1084	420	164
Clock Mux (Sync, $k = 3$)	911	82	80	200
Divider (By 2)	986	282	84	200
Divider (By 3)	2527	793	294	153
Divider Optimized	40	67	0	200

Table 6.3: TU resource utilization when targeting a Xilinx Ultrascale+ device (VU9P). We constrained our designs to 200 MHz as that represents a substantial margin over typically realizable f_{fpga}.

6.14 FPGA QoR And Scalability

The feasibility of our approach hinges on the assumption that managing large timestamps for relatively few and small TUs would incur a small resource cost relative to the rest of the simulator. Here we quantify these costs by measuring TU and hub-unit resource utilization and reporting their f_{max}.

6.14.1 TU Resource Utilization

One would expect that building TUs that process multiple streams of messages with large timestamps would bring about a large resource cost despite the underlying simplicity of the target circuits they model. To collect these results for a given TU, we elaborate a top-level wrapper circuit that registers all input and output interfaces of the TU. We then used Vivado version 2018.2, with the default synthesis and implementation strategies, to measure circuit QoR. To estimate f_{max} we over constrained the host clock in these designs to 200 MHz.

Table 6.3 lays bare the reality that the abstraction cost of the library primitives used in the baseline implementations is significant. The largest driver of utilization, in both optimized and baseline TUs, are timestamp comparisons. These occur in multiple places in the baseline units, since each library primitive independently schedules across its local inputs. Conversely, in the optimized ICG and clock multiplexer, this happens only at the I/O boundary of the TU. Most starkly, since there is only a single input to our clock divider TU, no timestamp comparisons are required: this produces the large savings shown in Table 6.3. Nonetheless, as a fraction of the resources available on the VU9P, TUs resource utilization, even in the unoptimized models, is insignificant: the baseline clock mux uses 0.32% of the VU9P's available LUTs, whereas the optimized TU uses just 0.1%[1]. We suspect that using a

[1]This is still an amusingly large figure given that a fraction of a single LUT is required to implement most of the other 1-bit, 2:1 multiplexers in our simulators.

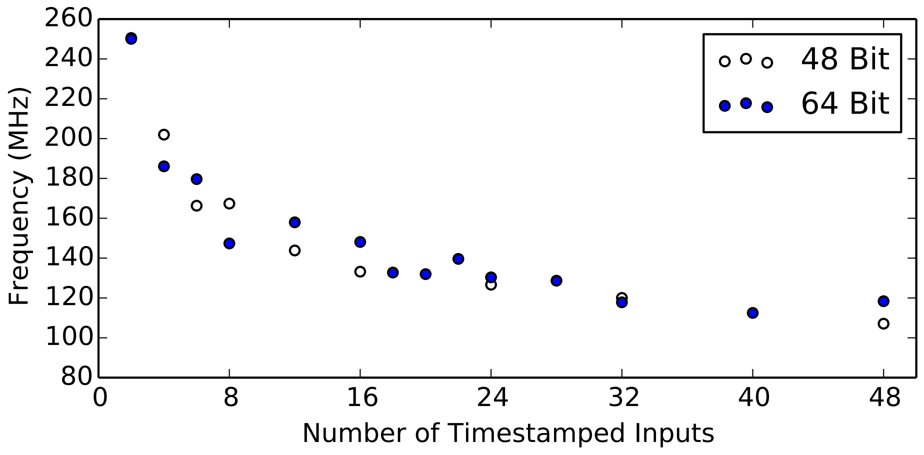


Figure 6.8: Measured f_{fpga} for simulators of 1-bit-wide scan-chain whose number of timestamped inputs is equal to the length of the chain plus one. Each point consists of a single design run. The underlying timing variability in FPGA implementation is evident with many 64-bit designs out-performing 48-bit equivalents.

more optimized message encoding and writing TU implementations that avoid acting directly on absolute timestamps would suffice to make the cost of the entire timestamped subgraph negligible in nearly all target systems. For example, if an upper bound between events is known *a priori*, a narrower internal timestamp may be used to disambiguate event orderings without having to operate on the full timestamp width.

TU f_{max} is also acceptably fast. While some of the baseline units fail to meet timing at 200 MHz, all of the optimized units do. 200 MHz is considerably greater than the achievable f_{fpga} of systems we typically simulate and thus is unlikely ever adversely affect f_{fpga}.

6.14.2 Hub Unit QoR and f_{max} Scaling

Another scaling challenge of our approach are costs associated with the frontend of the hub unit's pipeline, which features a large minimum-reduction across timestamps. We measured the per-timestamped-input scaling trends of the hub by synthesizing a scan chain where each register is asynchronously reset by an independent source. This adds a new timestamped input per register in the chain while only using a single BUFGCE. Here we build complete simulator bitstreams using FireSim's standard build flow with the "timing" strategy and Vivado version 2018.3.

In Figure 6.8, we plot f_{fpga} of the simulator as a function of the number of timestamped inputs used under both 48 and 64 bit timestamps. To produce these numbers, we coarsely overconstrained f_{fpga} (by up to 10 MHz in some cases). Given this heavy-handed overconstraint and the inherently stochastic nature of FPGA place and route especially when lacking placement constraints, fluctuations in f_{fpga} are expected. The overall trend is clear however.

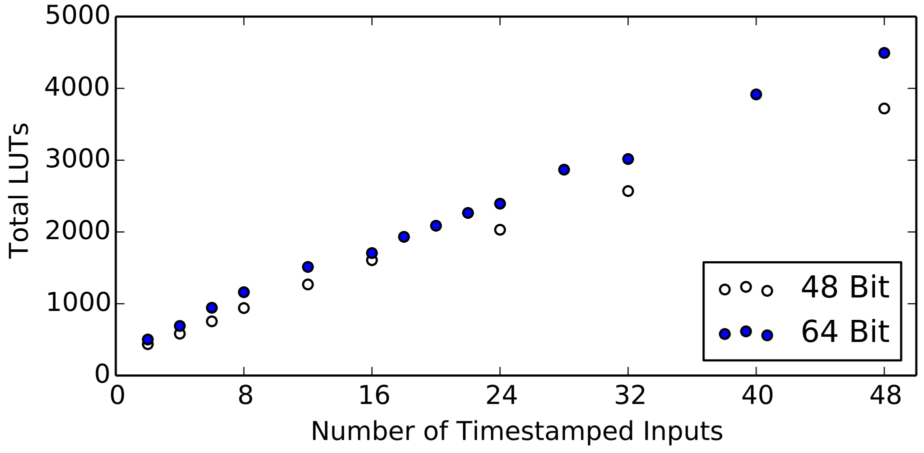


Figure 6.9: Measured LUT utilization, combined logic and memory, of the design hierarchy enclosed by the simulation wrapper as a function of timestamped input count. The main driver of increased utilization is the frontend of the hub unit's control pipeline, followed by the introduction of new message queues.

For both widths, Vivado can successfully close timing over 100 MHz when using as many as 48 timestamped inputs. While this is beyond the f_{fpga} we achieve for larger targets it may become problematic in highly congested designs. Practically speaking, global clock resource limitations will prevent the hub's frontend from becoming the simulators critical path. If we consider $N = 12$, the point at which Vivado struggles to place and route more global clocks, the simulators f_{fpga} was 144 MHz and 158 MHz for 48b and 64b timestamped respectively. By comparison, recall that with multi-cycle setup constraints the Rocket and Boom targets of the previous chapter closed timing at 150 MHz and 90 MHz respectively.

Another potential concern is the LUT utilization of the frontend of the hub. In Figure 6.9, we report LUT utilization of all modules under the simulation wrapper versus the length of the scan chain. This includes the hub and channel implementations, but excludes all bridges and FPGA-shell collateral like DRAM controllers. LUT utilization scales nearly linearly with increasing input count with approximately 86 and 74 additional LUTs required per timestamped input for 64 and 48 bit designs respectively. While not insignificant, the 4494 LUTs used in the 48 input, 64-bit variant accounts for just 0.38% of the total available LUTs in a Xilinx Ultrascale+ VU9P.

Chapter 7

Full-System Case Studies of SoCs Deploying Frequency Scaling

The support we have described thus far unlocks fast deterministic simulation of a large space of more realistic SoCs. To illustrate this, and demonstrate some of the compelling properties of our infrastructure, in this chapter we present two full-system experiments. In the first, we simulate a Rocket-based SoC that uses dynamic frequency scaling to stay within a desired thermal envelope. This serves as a simple example for how our system could be used to study novel thermal management and DVFS policies. In the second, we evaluate the simulation performance of a BOOM-based SoC when using our ICCAD2019 [70] register file optimization, to explore whether slowdowns induced by our PDES-based approach can be hidden by those introduced by multi-cycle optimizations.

7.1 Thermal Regulation via Dynamic Frequency Scaling

In this case study, we explore an SoC that, using Linux's standard thermal and frequency scaling subsystems, switches between two core frequencies to stay within a desired thermal envelope. Our infrastructure is unique among academic works in its generality and non-invasiveness: in principle no simulation-specific devices need to be exposed to target software, and target drivers need not be modified for FireSim since they continue to write to the same memory-mapped registers that would be present in the actual chip. Of course, performance-accurate simulation is only one half of this puzzle—a detailed power model is required to complete the loop. Prior work based on FireSim, notably Simmani [59], would be a good fit if and when it is upstreamed into FireSim. In lieu of a more accurate power model, we've used a simple replacement which is sufficient to demonstrate our simulation infrastructure.

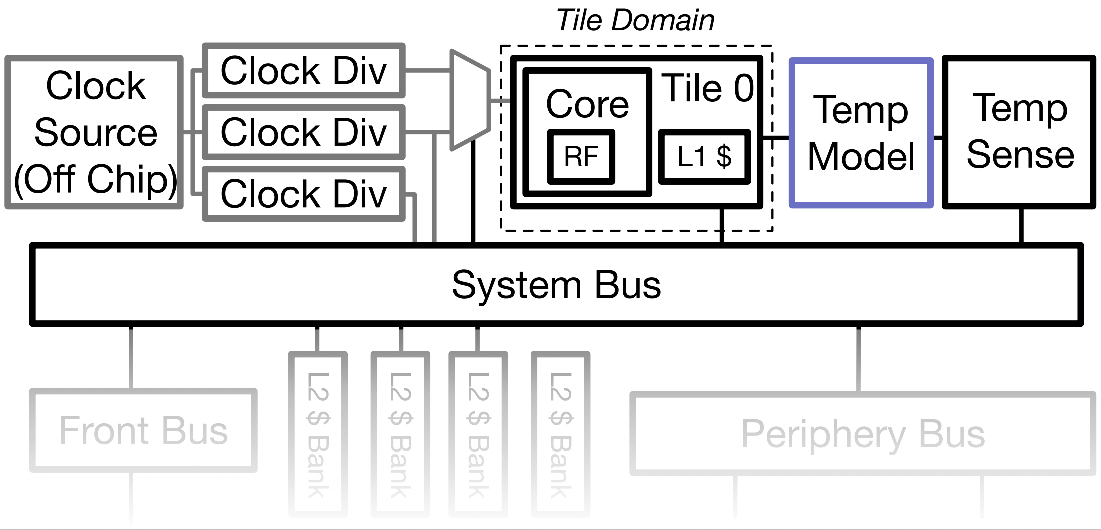


Figure 7.1: The modified, single-tile target used to illustrate thermal throttling. Modules in grey are TUs. The "Temp Model", in blue, is a CPU-hosted bridge. It is connected to the tile's trace interface to measure cycle count and instructions retired, and drives an 8-bit register in the "Temp Sense" module.

7.1.1 Target SoC Description

Our target is a single-core rocket-based SoC, shown in Figure 7.1, which differs minimally from what one can generate from Chipyard 1.4 out-of-the-box. While it would be easy to implement a simple PLL TU that could step up a slower reference clock to the required frequency, here we directly supply a high-speed reference clock that is divided down on chip. This matches what Chipyard does by default for software RTL simulation. We modified the divider sources to annotate themselves when they are instantiated (as in Listing 6.1), and instantiated our clock source bridge to supply the reference. Our largest clocking-related change was to introduce tile clock multiplexers which select between the uncore clock and a fast clock that runs at two times the uncore frequency. The select for this multiplexer is driven by a memory-mapped register bound to the periphery bus which resides in the uncore domain.

To integrate a temperature model, we added a temperature sensor that consists of an 8-bit memory-mapped register fed by a CPU-hosted temperature bridge. Target memory-mapped reads to this register return twice the current temperature in Celsius. The bridge periodically updates this register as a function of tile-cycles elapsed and instructions retired. To do so, the bridge driver polls its module every P uncore cycles, measures a change in cycle count (ΔC) and instructions retired (ΔI), calculates an updated temperature (T_{new}), and writes the updated temperature back to the target. The temperature model selects a core voltage by adding a scaling factor proportional to ΔC to a base. In the overwhelming majority of cases, V will be either be the base voltage or 1.5× greater, as the tile is unlikely to undergo a frequency change in any given polling interval. We give the temperature model used in the bridge below:

$$\begin{aligned} V &= 0.5 + \frac{\Delta C}{2P} \\ E_{dynamic} &\propto \Delta I * V^2 \\ E_{static} &\propto \Delta C * V \\ Q_{heating} &\propto E_{dynamic} + E_{static} \\ Q_{cooling} &\propto P(T_{ambient} - T_{old}) \\ T_{new} &= T_{old} + Q_{heating} + Q_{cooling} \end{aligned} \tag{7.1}$$

This simple model, while completely insufficient for conducting meaningful power studies, suffices for creating regimes of high dynamic power utilization which will induce Linux to switch between the two operating points.

7.1.2 System Software Description

Practically speaking, the most time consuming aspect of building this prototype was writing system software, since we could not reuse code from an existing chip. While Linux provides many generic drivers that can be configured entirely from the device tree, in the short term it was easier to write primitive drivers as kernel modules tailored for our target. We wrote two new kernel modules, requiring roughly 200 lines of mostly boilerplate C code:

- A frequency scaling driver (extending `cpufreq_driver` that binds to Linux's `cpufreq` subsystem. `cpufreq` is the canonical mechanism for implementing dynamic frequency scaling in Linux. The target-specific component of this driver supplies a table of two operating points and implements functions for initialization, de-initialization, and frequency switching. The latter accepts an index into the aforementioned table and writes to the tile clock multiplexer accordingly.
- A thermal zone device driver that implements a function to read the device's temperature. This driver defines polling intervals, temperature trip points and trip types. We configured the driver polls the temperature sensor every 100 ms and added a single trip point set to 40C.

Interaction between these two drivers is fairly straightforward. When Linux brings up the `cpufreq` subsystem during boot, it creates a new policy backed by our custom driver. We use Linux's `ondemand` governor for this policy, which picks the fastest available operating point unless the system is idle. Later in boot, the thermal zone device for our temperature sensor initializes and binds itself as a cooling device to the `cpufreq` policy above. When the thermal zone device reads a temperature above the 40C trip point, `cpufreq` will restrict the available operating points of the policy to the slower frequency. When this occurs the policy

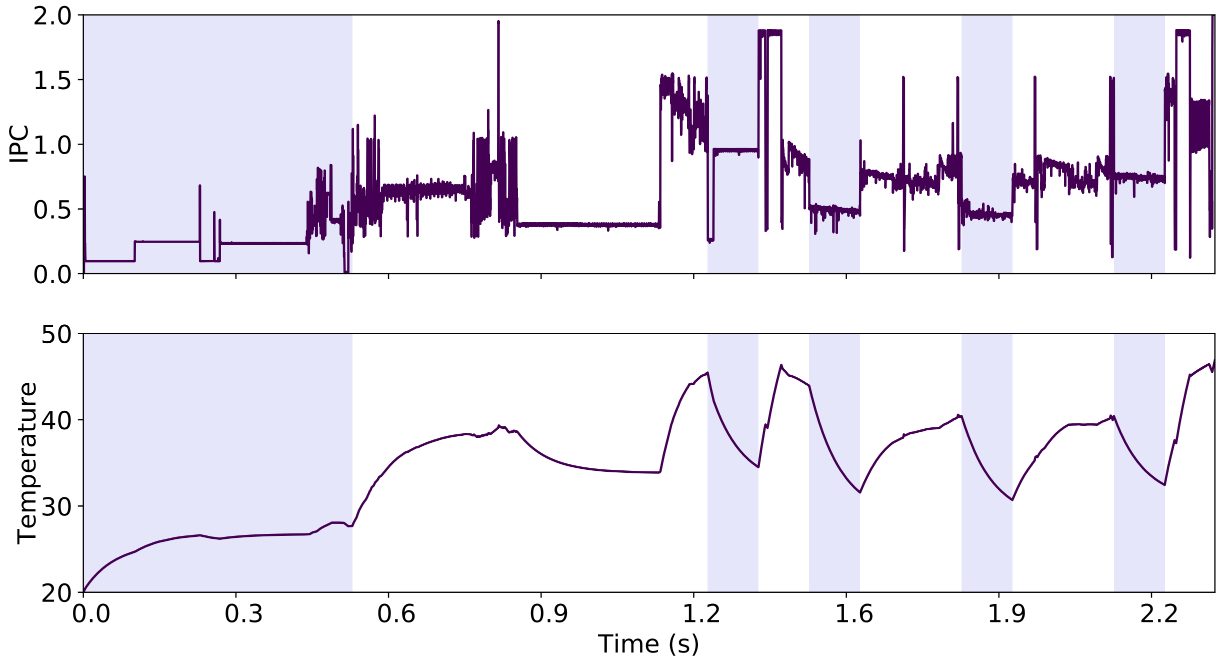


Figure 7.2: IPC and temperature (C) versus simulated time for `657.xz` running its test input. Regions in lavender indicate the core is running at the slower operating point (uncore frequency) whereas regions in white indicate is core is running at two times the uncore frequency. We plot IPC and temperature sampled every hundred thousand cycles, and apply a ten-sample rolling average.

is then forced to update: it switches to the available operating point, which produces a write to a memory-mapped register driving our clock multiplexer. When the temperature drops below this trip point, both operating points once again become available, and so unless the system is idle, the ondemand governor picks the faster operating point and the clock multiplexer setting is updated.

7.1.3 Demonstration

To demonstrate the operation of this system, we booted Linux on the target above, with our custom drivers, and ran the SPEC 2017 benchmark `657.xz` with its smaller "test" input. We tuned the coefficients of our model to produce frequency transitions in the timescales of this benchmark. Notably, we set $P = 10^6$, and $T_{ambient} = 20C$. In Figure 7.2, we plot temperature (C) and IPC versus time. We calculate IPC in terms of the fixed uncore clock to better highlight operating point changes: this permits rocket's IPC to reach two when running at the faster operating point despite being a scalar core. Regions of the graph shaded in lavender indicate the core is running at the slower operating point; the graph has white background otherwise.

After coming out of reset, the tile starts at the slower frequency. Here it remains until Linux brings up the `cpufreq` subsystem. After initialization, the governor switches to the faster operating point, producing the first transition visible in Figure 7.2. Later during Linux boot the thermal zone device for our temperature sensor is registered, and bound to the `cpufreq` policy above. Since the chip is not above 40 C, no frequency changes are made. Linux boot completes around 0.8 seconds into simulation time and the workload immediately starts.

The first throttling event occurs after a burst of relatively high IPC in the benchmark around 1.2 seconds. Here there is a precipitous drop in IPC that reflects that the core frequency has been halved. After one polling interval of the sensor, the temperature of the device has dropped sufficiently to re-enable the faster operating point, and the policy switches back to the faster mode. The policy performs this dance for the remainder of the simulation, though later periods of the benchmark have relatively lower IPC which permit the chip to run at the higher frequency longer before the core is throttled.

We ran this workload on an earlier version of our prototype that used the baseline TUs. We measured $FMR = 12.5$ giving an f_{emul} of 9.6 MHz. The increase of FMR over the 11.0 baseline we reported in Table 6.1 is due to simulation stalls induced by the temperature bridge to measure and update the core temperature. There is no reason this can not be overlapped with simulator execution, which would be critical if we re-ran this experiment using the optimized TUs. Indeed, with the optimized TUs we expect f_{emul} to be close to 60 MHz—a figure approaching the throughput achievable using prototyping-based approaches like that proposed by Mantovani et al. [72] but without many of the aforementioned limitations.

7.2 Latency Overlapping in Resource-Optimized BOOM-Based Simulators

One justification for exploring a distributed approach was that a potential FMR increase may overlap with one brought about by multi-cycle optimizations. This might hide some or all of the potential slowdown. These optimizations increase FMR significantly because tokens must propagate between a hub and spoke (the same effect responsible for two cycles of latency in our clock gate target) but, in contrast to some of our TUs, the UU itself takes multiple host cycles to process one target cycle. Theoretically, this means that FMR increases brought about by multi-cycle optimizations should exceed many of the figures we reported in Section 6.12 and Section 6.13.

To illustrate this effect we measure the performance of three quad-core BOOM simulators, one based off mainline FireSim and two using the work of this chapter. We use the "LargeBoom" configuration: core microarchitectural parameters match those previously shown in Table 5.1. As we reported in our ICCAD paper [70], this system cannot fit on a single VU9P device, so we will recruit the assistance of the multi-cycle RAM optimization to efficiently implement BOOM's integer and floating point register files. Note, the target

		Tiles @ 1.5 GHz		Tiles @ 750 MHz	
	Static	**Baseline**	**Optimized**	**Baseline**	**Optimized**
FMR	12.1	21.0	12.1	16.0	7.0
FMR$_{tile}$				32.0	14.0

Table 7.1: Measured FMR of a quad-core multi-clock"Large BOOM" SoC with the multi-cycle RAM optimization applied to all register files. Tile frequency in the dynamically clocked configurations (columns labeled with "Tiles @ {1.5 GHz, 750 MHz}") was set by the first stage bootloader and fixed for the remainder of simulation. FMR in these systems is with respect to the 1.5 GHz clock regardless of frequency selection. FMR$_{tile}$ reports the FMR of the clock used to drive the tiles.

RTL for the statically and dynamically clocked systems differs only in how their clocks are generated. All of these systems have clocks running at three distinct frequencies: a 500 MHz DRAM clock, a 750 MHz uncore clock, and 1.5 GHz tile clock. The statically clocked system uses the clock generation scheme described in the previous chapter and fixes all four tiles at 1.5 GHz. The two dynamically clocked systems resemble the target we used in the thermal throttling demonstration (Figure 7.1: a high-speed clock source (1.5 GHz) feeds parallel clock dividers to provide the uncore and DRAM clocks, and each of the four tiles is given its own clock multiplexer. The two dynamically clocked variants differ only in their TU selection: ones uses the lookahead-optimized units and the other uses the baseline implementations. All-in-all, resource utilization is approximately the same across all designs, though the dynamically clocked configurations use seven BUFGCEs instead of three. All three designs closed timing at 40 MHz.

For the pair of dynamically clocked systems, we consider two runtime operating conditions: one in which all cores are clocked at 1.5 GHz, and one in which they are all clocked at 750 MHz for the duration of boot. We had the first-stage bootloader (BBL) configure the tile multiplexers before the kernel starts. In the slower case, the RAM models are only active in every other fast-clock cycle, providing fewer opportunities for the delays in timestamped message passing to be hidden. As a final note, in this experiment we use the RAM optimization first presented at ICCAD2019 [70] and not the throughput-optimized variant presented by A. Magyar in his dissertation [71]. This unit statically schedules all register file accesses thus has a fixed FMR. A. Magyar reports an FMR of 12.1 for "Large BOOM" configurations using this optimization.

To a first order, we expect latency overlapping to occur when two conditions are met:

1. Latency-critical messages propagate in the same simulator timestep where a multi-cycle UU is active.

2. The data values of the aforementioned messages do not combinationally depend on tokens generated by the UU with the longest latency.

Since the register driving the clock multiplexer select is many cycles removed from a register file access, the second condition is met in our designs. Otherwise, the FMR penalties of register file access and message propagation would be serialized. Thus, any observed slowdowns should arise because the first condition has not been met.

We report the FMR of these configurations in Table 7.1. Since the timestamped subgraph in these simulators is identical to that of the clock multiplexer organization we studied in Section 6.12, we would expect FMRs of at least 11.0 and 2.0 for designs using baseline and optimized TU implementations respectively. When running at the fast clock frequency, the optimized clock multiplexer configuration can completely hide its FMR cost because the negative edge is squeezed out of hub-unit pipeline while the multi-cycle RAM model is operating (this is possible because the hub already has a future positive edge it can schedule). Conversely, in the baseline configuration the negative edge timestep cannot be squeezed out, since the clock multiplexer requires a null message at that timestamp to produce the next positive-edged message on the multiplexer-selected clock. Since the RAM model is not active on negative edges of the tile's clock, this latency cannot be overlapped. This produces the apparent partial overlapping observed in the baseline configuration.

When all tiles run at the slower frequency, FMR decreases but FMR_{tile} increases in both cases, because the RAM models remain idle on every other fast clock edge. Note that FMR_{tile} is larger than a static configuration would be running at the same frequency (where $\text{FMR} = 12.1$). Here the FMR of the timestamped subgraph is exposed directly to the simulator on every other fast clock positive edge. Thus, adding the expected FMR of the the timestamped subgraphs running in isolation (11.0, 2.0), to the measured FMRs of the "Tiles @ 1.5 GHz" case, produces the observed FMR_{tile} in these slower configurations.

The main takeaway here is that, if a TU depends only on messages in timestamps in which multi-cycle UUs are active, their latencies can generally be completely hidden. For example, if we were to add clock gates to the tile, the four cycle FMR penalty would be hidden so long as at least one other core is not clock gated. In a single-tile configuration, any cycle in which the core is clock gated disables execution of the multi-cycle RAM unit, exposing the clock-gate's FMR penalty of four. The expected FMR of this system would therefore be $\text{FMR} = 4p + 12.1(1-p)$, where p is the fraction of tile clock cycles in which the clock is gated. Naturally, these observations are closely tied to our current implementation, specifically to that of the hub unit—in the next chapter, we explore future work including some ideas that would expose more opportunities to overlap latencies.

Chapter 8

Limitations and Future Work

Perhaps the most attractive aspect of our PDES-based approach is that it provides a flexible abstraction for describing arbitrary models of non-SSM circuits. While it is more resource intensive then a more direct centralized approach, the real cost of this abstraction is increased FMR. As we showed in Section 7.2, in some cases the latency introduced by modeling TUs can be overlapped with the increased execution time of multi-cycle UUs—in this way, our approach dovetails with Golden Gate's defining feature. However, the overwhelming majority of FireSim users simulate smaller target systems in which no optimizations are required and the expectation is that simulators run close to unity FMR. So, for our approach to be viable, it needs to recoup the performance losses it has introduced over the implementation described in Chapter 5. This will require overcoming throughput bounds that limit feedforward topologies (e.g., clock divider trees), and more critically, the latency bounds introduced by cycles in the timestamped subgraph.

8.1 Performance: Overcoming Throughput Bounds

To replace the existing clock generation in FireSim, described in Chapter 5, overcoming the throughput bounds of feedforward networks would be sufficient, as networks of clock sources and dividers can subsume the existing feature set.

The best case FMR of our approach is currently two because positive and negative edges for clocks are enqueued and dequeued as separate messages. There are a few potential fixes for this. The first is to use a better clock-message encoding, for example, one that encodes both positive and negative edge times. This is not a panacea, because both positive and negative edge times may not be known at enqueue time. In practice, TUs would need to support both single and double-edged variants of clock messages. Alternatively, for fixed clocks whose frequencies are known at the start of simulation, the time zero message could simply encode the period and phase of the clock: sinks would need only one message to be set for the remainder of simulation.

Even if the FMR-of-two limitation is lifted, the presence of high-frequency reference clocks, that are not used to drive logic in the target, can introduce a new bandwidth bound. In a system with clocks running at 1.5 GHz, 1 GHz, and 750 MHz (as in our system described in Chapter 5), using only integer dividers requires a 3 GHz reference clock. Even with the denser double-edged message encoding described previously, the system would run with FMR of two for the core domain. To improve this and support modeling high-frequency clock generators like VCOs in *independent* TUs, some form of multi-period encoding of clocks is unavoidable.

Without this there are two simple paths forward. The first is to preclude modeling high-frequency clock sources and downstream dividers as distinct units, but instead fold them into a single unit. This removes the high frequency channel from the timestamped subgraph. Since output dividers are often integrated into PLL IP, it would be natural to swap them out together and replace them with a general but abstract PLL model whose implementation is optimized to ensure high output-message throughput. In the short term, if we do not wish to modify clock token encodings, we could configure clock dividers to assume their inputs have fixed periods, which they would learn after receiving their first pair of input messages. This would permit them to look arbitrarily far ahead of their inputs.

8.2 Performance: Overcoming Latency Bounds

If there is a fundamental problem with the deadlock-avoiding approach we have presented, it is that latency through loops in the timestamped subgraph graph can substantially increase FMR. This is especially evident in the baseline TUs. Since many clocking primitives are tightly coupled to the rest of the target, it not always possible to build TUs that have sufficient lookahead to hide message transmission and hub-unit pipeline latency. This was most clearly demonstrated with our optimized ICG, which directly exposes the four-cycle hub-TU loop latency. Staying within the regime of using decoupled TUs, there are a few general approaches to improving FMR.

This first is to simply cut latencies where possible. The four-cycle update loop between the hub and a TU could be improved by using flow-through queues between the hub unit and sink TU, shaving one cycle of latency. More aggressively, it may also be possible to combine the first two stages of the hub's control pipeline for small numbers of timestamped inputs. Since Vivado places all BUFGCEs in the same clock region as the main simulator clock (to produce balanced trees), there may be little benefit to having the extra stage as there is insignificant routing delay from the stage-1 registers to the clock buffers. Both of these "fixes" have the potential to decrease f_{fpga}, and may not be universally applicable.

A more intelligent messaging encoding and channel implementation is also worth exploring. Currently, null messages occupy slots in the channel queues. This may produce head-of-line blocking where a TU is forced to dequeue a null message before it can observe a real transition, potentially adding an extra cycle of latency. Ideally, these extraneous null messages would be squashed out. In practice there is no need to enqueue null messages, instead, the channel could update a secondary time register whose value would always match or exceed the timestamp of the non-null message at the head of the queue. When any new message is enqueued this value is increased. Null messages could then be dropped with no loss of information.

Some other approaches which we have not previously mentioned in earlier in sections include:

- Introducing global optimizations that would share information about source behavior to sinks. The example from the previous section where a clock divider may assume its input is fixed is one example of this.

- Allowing hub-unit output channels to decouple. If a clock domain has just fired, timestamped outputs affected by transitions in that domain likely are not going to change in the near future. This information could be captured in first output message, instead of waiting for additional timesteps to issue new null messages.

8.3 Future Work, Performance Aside

Working within the PDES abstraction described in this dissertation there are many other avenues, beyond performance optimizations, worthy of consideration:

- Build out a larger library of optimized TUs. Notably absent from this work is a PLL TU. Here, it would be relatively easy to build a simple model with integrated output dividers. This could be expanded to include a laundry list of other features, including modeling locking time and supporting fractional output dividers. As previously mentioned, all of this could be done without exposing the high-frequency VCO output to the simulation graph.

- Add asynchronous reset to existing TUs. For this to perform well, the asynchronous-reset message stream will need to lead other inputs (which may be difficult to ensure), or the TU will have to find asynchronous-reset relative lookahead, which generally speaking is not possible. Here the designer could artificially introduce lookahead if they know that asynchronous-reset-launched transitions can be safely be ignored such that they may be delayed.

- Explore automatic translation of Verilog to TU implementations. Writing complex TUs can be difficult. This would sidestep that challenge and produce models that, ideally, faithfully represent the source (once the tool itself has been verified). For this to work well, the compiler must find lookahead opportunities that span the circuit.

- If bespoke TUs are required for high performance having a better strategy for verifying them against a reference becomes crucial. For the same reasons LIME was effective for verifying primitive LI-BDNs, an equivalent model checking flow would be useful for

TUs. Simulator deadlock avoidance could be guaranteed by proving a TU implementation has non-zero lookahead. Checking message correctness would be be considerably more difficult.

- Integrate an accurate power model. In providing RTL-faithful timing accuracy, our infrastructure satisifies one critical prerequisite for performing accurate research on DVFS-capable systems. Adding a power model, tailored for the target, would make FireSim a compelling framework for full-stack DVFS research. Here, integrating Simmani [59] is the obvious next step. This would only require a new target transformation, to wire up signals of interest as selected by Simmani to a custom bridge, and no internal modifications to Golden Gate. Alternatively, a snapshot and replay approach like Strober's [58] is also tractable but considerably more challenging to enable under Golden Gate due to increased simulator complexity.

Chapter 9

Conclusion

Our goal for the FireSim project was to build a radically inexpensive, yet fast and productive, cycle-accurate full-system simulation technology. This was in service of reducing a key contributor to the NRE of building silicon. While FireSim is most similar to existing hardware emulators, our approach is unique in that it uses a single commercial-off-the-shelf FPGA, and depends on a completely open-source toolchain. However, it was difficult to make a comparison to existing hardware emulation solutions in good faith, as early versions of FireSim had two critical limitations: the SoCs it could support were small, and could possess only a single-clock domain.

We designed Golden Gate to address these limitations (Chapter 4). First, Golden Gate uses a LI-BDN target formalism and RAMP-inspired multi-cycle optimizations to fit larger SoCs on a single FPGA. Descriptions of these optimizations can be found in Albert Magyar's dissertation [71]. Overcoming the clocking limitations was the primary focus of this dissertation. To model multiple clock domains, we introduced a new FAME transform that avoids using dedicated FPGA clocking resources like prior academic work [72], in favor of a clock-gating scheme that should be portable to many different FPGA platforms (Chapter 5). To model clock switching and generation circuits, which form the basis of dynamic frequency scaling support in realistic SoCs, we introduced a timestamped subgraph into the simulator that implements a conservative PDES (Chapter 6).

One take-away from our work in supporting clock switching and generation structures is that closely coupled circuits, like ICGs and clock multiplexers, are probably best simulated *in situ*, instead of being extracted into a decoupled unit. State for these circuits can be left in the target, and combinational functions they perform on clocks can be hoisted into the hub unit's second stage. Here they would act directly on future clock enables to drive additional clock buffers. Clearly, this would be insufficient for modeling clock generators, like PLLs, that can't be described as combinational functions on existing clocks. For modeling circuits of this nature having an independent TUs seems entirely sensible. This hybrid approach would also reduce the number of timestamped inputs on the hub unit, addressing a potential scalability challenge in systems with many clocks. Implementing and studying this approach is the logical continuation of the work described in this dissertation.

While Golden Gate has made inroads in simulating far more realistic SoCs on a single FPGA, there are still many domains under the FireSim project that require attention. First, FireSim should be ported to other FPGAs to verify our claims about the flexibility of our approaches. Here there are ongoing efforts both at Berkeley and from the community at large. Perhaps the biggest frontier for innovation lies in improving FireSim's debuggability. Snapshotting features, to provide greater visibility over the target, are important feature in commercial hardware emulators that FireSim currently lacks. MIDAS did have support for this, but its implementation presupposed a monolithic, single clock domain hub. Supporting resource-efficient state capture that co-exists with resource optimizations is both an important and compelling avenue for future work.

FireSim's growing user base, both academic and industrial, suggests our vision for a more cost-effective full-system simulation technology addresses a material technology gap. We hope that FireSim and the contributions of this dissertation inspire more academic research in this domain in the future.

Bibliography

[1] Defense Advanced Research Projects Agency. *DARPA Announces First Bug Bounty Program to Hack SSITH Hardware Defenses*. https://github.com/firesim/firesim/issues/613. 2020 (Accessed July 29th, 2020).

[2] Amazon. *Amazon EC2 F1 Instances (Preview)*. https://aws.amazon.com/ec2/instance-types/f1/. 2016.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21.

[4] Dough Amos, Austin Lesea, and Ren Richter. *FPGA-Based Prototyping Methodology Manual. Best Practises in Design-For-Prototyping*. Synopsys Press, 2011. ISBN: 978-1617300042.

[5] Hari Angepat, Dam Sunwoo, and Derek Chiou. "RAMP-White: An FPGA-Based Coherent Shared Memory Parallel Computer Emulator". In: *8th Annual Austin CAS Conference*. Vol. 26. 2007.

[6] *Apple Unleashed M1*. https://www.apple.com/in/newsroom/2020/11/apple-unleashes-m1/. 2020 (Accessed March 29th, 2021).

[7] Krste Asanović. "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers". In: Santa Clara, CA: USENIX Association, Feb. 2014.

[8] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016.

[9] *AWS Graviton Processor*. https://aws.amazon.com/ec2/graviton/. 2020 (Accessed March 29th, 2021).

[10] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Zimi Hanono, David M Hoki, and Anant Agarwal. "Logic Emulation With Virtual Wires". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.6 (1997), pp. 609–626.

[11] Jonathan Bachrach, Albert Magyar, Palmer Dabbelt, Patrick Li, Richard Lin, and Krste Asanović. "Cyclist: Accelerating Hardware Development". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 1011–1018.

[12] 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. San Francisco, California: ACM, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: `10.1145/2228360.2228584`. URL: `http://doi.acm.org/10.1145/2228360.2228584`.

[13] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. "Computer and Redundancy Solution for the Full Self-Driving Computer". In: Aug. 2019, pp. 1–22. DOI: `10.1109/HOTCHIPS.2019.8875645`.

[14] Luiz Andre Barroso, Sasan Iman, M Dubois, and K Ramamurthy. "RPM: A Rapid Prototyping Engine for Multiprocessor Systems". In: *Computer* 28.2 (1995), pp. 26–34.

[15] Daniel K Beece, G Deiberg, Georgina Papp, and Frank Villante. "The IBM Engineering Verification Engine". In: *25th ACM/IEEE, Design Automation Conference. Proceedings 1988*. IEEE. 1988, pp. 218–224.

[16] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanović. "FASED: FPGA-Accelerated Simulation and Evaluation of DRAM". In: *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. FPGA '19. Seaside, CA, USA: ACM, 2019. ISBN: 978-1-4503-6137-8.

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`. URL: `http://doi.acm.org/10.1145/2024716.2024718`.

[18] Randy E. Bryant. *Simulation Of Packet Communication Architecture Computer Systems*. Tech. rep. USA, 1977.

[19] Cadence. *Cadence Launches Protium X1, the First Scalable, Data Center-Optimized Enterprise Prototyping System for Early Software Development.* `https://www.cadence.com/en_US/home/company/newsroom/press-releases/pr/2019/cadence-launches-protium-x1--the-first-scalable--data-center-opt.html`. 2019 (Accessed January 4th, 2021).

[20] Cadence. *Cadence Palladium Z1 Enterprise Emulation Platform.* `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-z1-ds.pdf`. 2015 (Accessed January 4th, 2021).

[21] Luca Carloni, Kenneth McMillan, and Alberto Sangiovanni-Vincentelli. "Theory of Latency-Insensitive Design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.9 (Nov. 2006), pp. 1059–1076. ISSN: 0278-0070. DOI: `10.1109/43.945302`. URL: `http://dx.doi.org/10.1109/43.945302`.

[22] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Tech. rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, June 2015.

[23] K. Mani Chandy and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". In: *IEEE Transactions on Software Engineering* SE-5.5 (1979), pp. 440–452.

[24] Chen Chang, John Wawrzynek, and Robert W Brodersen. "BEE2: A High-End Reconfigurable Computing System". In: *IEEE Design Test of Computers* 22.2 (Mar. 2005), pp. 114–125. ISSN: 0740-7475. DOI: `10.1109/MDT.2005.30`.

[25] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers.* 2016, 262–263.

[26] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007).* IEEE. 2007, pp. 249–261.

[27] Niket K Choudhary, Salil V Wadhavkar, Tanmay A Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H Dwiel, Sandeep Navada, Hashem H Najaf-abadi, and Eric Rotenberg. "Fabscalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template". In: *ACM SIGARCH Computer Architecture News.* Vol. 39. 3. ACM. 2011, pp. 11–22.

[28] Eric S Chung, Eriko Nurvitadhi, James C Hoe, Babak Falsafi, and Ken Mai. “A Complexity-Effective Architecture for Accelerating Full-system Multiprocessor Simulations Using FPGAs”. In: *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. FPGA ’08. Monterey, California, USA: ACM, 2008, pp. 77–86. ISBN: 978-1-59593-934-0. DOI: `10.1145/1344671.1344684`. URL: `http://doi.acm.org/10.1145/1344671.1344684`.

[29] Henry Cook, Wesley Terpstra, and Yunsup Lee. “Diplomatic Design Patterns: A TileLink Case Study”. In: *CARRV ’17*. 2017.

[30] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. “The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips”. In: *IEEE Micro* 38.2 (2018), pp. 30–41. DOI: `10.1109/MM.2018.022071133`.

[31] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. “Design of Ion-Implanted MOSFET’s With Very Small Physical Dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: `10.1109/JSSC.1974.1050511`.

[32] Monty M Denneau. “The Yorktown Simulation Engine”. In: *19th Design Automation Conference*. IEEE. 1982, pp. 55–59.

[33] Brandon H Dwiel, Niket K Choudhary, and Eric Rotenberg. “FPGA Modeling of Diverse Superscalar Processors”. In: *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*. ISPASS ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 188–199. ISBN: 978-1-4673-1143-4. DOI: `10.1109/ISPASS.2012.6189225`. URL: `http://dx.doi.org/10.1109/ISPASS.2012.6189225`.

[34] EETimes. *Quickturn’s New Emulation Family Sets the Standard for Verification Performance and Capacity*. https://www.eetimes.com/quickturns-new-emulation-family-sets-the-standard-for-verification-performance-and-capacity/. Jan. 1997 (Accessed July 5th, 2020).

[35] *Expanding Domestic Manufacturing of Secure, Custom Chips for Defense Needs*. `https://www.darpa.mil/news-events/2021-03-18`. March 18th, 2021 (Accessed March 29th, 2021).

[36] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. “Leveraging Latency-Insensitivity to Ease Multiple FPGA Design”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’12. Monterey, California, USA: ACM, 2012, pp. 175–

184. ISBN: 978-1-4503-1155-7. DOI: `10.1145/2145694.2145725`. URL: `http://doi.acm.org/10.1145/2145694.2145725`.

[37] Richard M. Fujimoto. "Parallel Discrete Event Simulation". In: 33.10 (1990). ISSN: 0001-0782. DOI: `10.1145/84537.84545`. URL: `https://doi.org/10.1145/84537.84545`.

[38] Richard M. Fujimoto, Rajive Bagrodia, Randal E. Bryant, K. Mani Chandy, David Jefferson, Jayadev Misra, David Nicol, and Brian Unger. "Parallel Discrete Event Simulation: The Making of a Field". In: *2017 Winter Simulation Conference (WSC)*. 2017, pp. 262–291.

[39] Greg Gibeling, Andrew Schultz, and Krste Asanović. "The RAMP Architecture & Description Language". In: *In 2nd Workshop on Architecture Research using FPGA Platforms*. 2006.

[40] GitHub. *Block Inclusive Cache SiFive*. `https://github.com/sifive/block-inclusivecache-sifive`. 2021 (Accessed February 1st, 2021).

[41] GitHub. *CCBench*. `https://github.com/ucb-bar/ccbench`. 2021 (Accessed February 1st, 2021).

[42] GitHub. *FireSim Github Issue 613 (Support Ticket for FETT)*. `https://www.darpa.mil/news-events/2020-06-08a`. 2020 (Accessed July 29th, 2020).

[43] GitHub. *Speckle*. `https://github.com/ccelio/Speckle`. 2021 (Accessed January 20th, 2021).

[44] Darryl Gove. "CPU2006 Working Set Size". In: *SIGARCH Comput. Archit. News* 35.1 (Mar. 2007), pp. 90–96. ISSN: 0163-5964. DOI: `10.1145/1241601.1241619`. URL: `https://doi.org/10.1145/1241601.1241619`.

[45] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. "Sources of Error in Full-System Simulation". In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 13–22.

[46] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: `10.1109/MICRO.2016.7783759`.

[47] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: `10.1145/1186736.1186737`. URL: `http://doi.acm.org/10.1145/1186736.1186737`.

[48] James C Hoe, Doug Burger, Joel Emer, Derek Chiou, Resit Sendag, and Joshua Yi. "The Future of Architectural Simulation". In: *IEEE Micro* 30.3 (May 2010), pp. 8–18. ISSN: 0272-1732. DOI: `10.1109/MM.2010.56`.

[49] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and Jonathan Wawrzynek. “Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942048.

[50] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. “Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations”. In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD ’17. Irvine, California: IEEE Press, 2017, pp. 209–216.

[51] David R. Jefferson. “Virtual Time”. In: *ACM Trans. Program. Lang. Syst.* 7.3 (July 1985), pp. 404–425. ISSN: 0164-0925. DOI: 10.1145/3916.3988. URL: https://doi.org/10.1145/3916.3988.

[52] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. URL: https://doi.org/10.1145/3140659.3080246.

[53] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. “Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 9781538659847. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[54] Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. “FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 715–731. ISBN: 9781450371025. DOI: 10.1145/3373376.3378455. URL: https://doi.org/10.1145/3373376.3378455.

[55] Donggyu Kim. “FPGA-Accelerated Evaluation and Verification of RTL Designs”. PhD thesis. EECS Department, University of California, Berkeley, May 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-57.html.

[56] Donggyu Kim, Christopher Celio, David Biancolin, Jonathan Bachrach, and Krste Asanović. “Evaluation of RISC-V RTL with FPGA-Acclerated Simulation”. In: *CARRV ’17*. 2017.

[57] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanović. “DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles”. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 76–764.

[58] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. “Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 128–139. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.21. URL: https://doi.org/10.1109/ISCA.2016.21.

[59] Donggyu Kim, Jerry Zhao, Jonathan Bachrach, and Krste Asanović. “Simmani: Runtime Power Modeling for Arbitrary RTL with Automatic Signal Selection”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1050–1062. ISBN: 9781450369381. DOI: 10.1145/3352460.3358322. URL: https://doi.org/10.1145/3352460.3358322.

[60] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. “RAMP Blue: A Message-Passing Manycore System in FPGAs”. In: *2007 International Conference on Field Programmable Logic and Applications*. Aug. 2007, pp. 54–61. DOI: 10.1109/FPL.2007.4380625.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[62] E Kronstadt and G Pfister. “Software Support for the Yorktown Simulation Engine”. In: *19th Design Automation Conference*. IEEE. 1982, pp. 60–64.

[63] Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 203–215. ISSN: 0278-0070. DOI: `10.1109/TCAD.2006.884574`.

[64] Charles Eric Laforest, Ming G. Liu, Emma Rae Rapati, and J. Gregory Steffan. "Multi-ported Memories for FPGAs via XOR". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '12. Monterey, California, USA: ACM, 2012, pp. 209–218. ISBN: 978-1-4503-1155-7. DOI: `10.1145/2145694.2145730`. URL: `http://doi.acm.org/10.1145/2145694.2145730`.

[65] Charles Eric LaForest and J. Gregory Steffan. "Efficient Multi-ported Memories for FPGAs". In: *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '10. Monterey, California, USA: ACM, 2010, pp. 41–50. ISBN: 978-1-60558-911-4. DOI: `10.1145/1723112.1723122`. URL: `http://doi.acm.org/10.1145/1723112.1723122`.

[66] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1*. Tech. rep. UCB/EECS-2015-262. EECS Department, University of California, Berkeley, Dec. 2015.

[67] Yunsup Lee and Andrew Waterman. "Managing Chip Design Complexity in the Domain-Specific SoC Era". In: *2020 IEEE Symposium on VLSI Circuits*. 2020, pp. 1–2. DOI: `10.1109/VLSICircuits18222.2020.9162812`.

[68] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Rimas Avizienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Bora Nikolic, and Krste Asanović. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro* 36.2 (2016), pp. 8–20. DOI: `10.1109/MM.2016.11`.

[69] Boris D. Lubachevsky. "Efficient Parallel Simulations of Dynamic Ising Spin Systems". In: *J. Comput. Phys.* 75.1 (Mar. 1988), pp. 103–122. ISSN: 0021-9991. DOI: `10.1016/0021-9991(88)90101-5`. URL: `https://doi.org/10.1016/0021-9991(88)90101-5`.

[70] Albert Magyar, David Biancolin, Jack Koenig, Sanjit Seshia, Jonathan Bachrach, and Krste Asanović. "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8.

[71] Albert Forte Magyar. "Improving FPGA Simulation Capacity with Automatic Resource Multi-Threading". PhD thesis. EECS Department, University of California, Berkeley, May 2021.

[72] Paolo Mantovani, Emilio G. Cota, Kevin Tien, Christian Pilato, Giuseppe Di Guglielmo, Ken Shepard, and Luca P. Carloni. “An FPGA-Based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems”. In: *2016 53nd ACM/EDAC/ IEEE Design Automation Conference (DAC)*. Austin, TX, USA: IEEE Press, 2016, pp. 1–6. DOI: `10.1145/2897937.2897984`. URL: `https://doi.org/10.1145/2897937.2897984`.

[73] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. “Graphite: A Distributed Parallel Simulator for Multicores”. In: *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE. 2010, pp. 1–12.

[74] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965).

[75] David M. Nicol. “Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks”. In: *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*. PPEALS ’88. New Haven, Connecticut, USA: Association for Computing Machinery, 1988, pp. 124–137. ISBN: 0897912764. DOI: `10.1145/62115.62128`. URL: `https://doi.acm.org/10.1145/62115.62128`.

[76] Edward J. Nowak. “Maintaining the Benefits of CMOS Scaling When Scaling Bogs Down”. In: *IBM Journal of Research and Development* 46.2.3 (2002), pp. 169–180. DOI: `10.1147/rd.462.0169`.

[77] Koray Öner, Luiz A Barroso, Sasan Iman, Jaeheon Jeong, Krishnan Ramamurthy, and Michel Dubois. “The Design of RPM: An FPGA-Based Multiprocessor Emulator”. In: *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*. 1995, pp. 60–66.

[78] Avadh Patel, Furat Afram, and Shunfei Chen. “MARSSx86: A Full System Simulator for x86 CPUs”. In: *DAC*. 2011.

[79] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel Emer. “HAsim: FPGA-Based High-detail Multicore Simulation Using Time-division Multiplexing”. In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. HPCA ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 406–417. ISBN: 978-1-4244-9432-3. URL: `http://dl.acm.org/citation.cfm?id=2014698.2014876`.

[80] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, and Joel Emer. “A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs”. In: *ACM Trans. Reconfigurable Technol. Syst.* 2.3 (Sept. 2009), 16:1–16:26. ISSN: 1936-7406. DOI: `10.1145/1575774.1575775`. URL: `http://doi.acm.org/10.1145/1575774.1575775`.

[81] Nathan Pemberton and Alon Amid. *FireMarshal: Reproducible Software Workload Management.* Mar. 2021.

[82] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs". In: *IEEE Micro* 40.4 (2020), pp. 93–102. DOI: 10.1109/MM.2020.2996145.

[83] Carl Pixley. "A theory and implementation of sequential hardware equivalence". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.12 (1992), pp. 1469–1478. DOI: 10.1109/43.180261.

[84] *QEMU: The FAST! Processor Emulator.* https://www.qemu.org. 2020 (Accessed August 2nd, 2020).

[85] Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. "PDES-A: Accelerators for Parallel Discrete Event Simulation Implemented on FPGAs". In: *ACM Trans. Model. Comput. Simul.* 29.2 (Apr. 2019). ISSN: 1049-3301. DOI: 10.1145/3302259. URL: https://doi.org/10.1145/3302259.

[86] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers". In: *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* SIGMETRICS '93. Santa Clara, California, USA: Association for Computing Machinery, 1993, pp. 48–60. ISBN: 0897915801. DOI: 10.1145/166955.166979. URL: https://doi.org/10.1145/166955.166979.

[87] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. "PULP: A Parallel Ultra Low Power Platform for Next Generation IoT Applications". In: *2015 IEEE Hot Chips 27 Symposium (HCS).* 2015, pp. 1–39. DOI: 10.1109/HOTCHIPS.2015.7477325.

[88] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, et al. "Intel Nehalem Processor Core Made FPGA Synthesizable". In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays.* ACM. 2010, pp. 3–12.

[89] Sanjit Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE).* Oct. 2018, pp. 1–10. DOI: 10.1109/MEMCOD.2018.8556946.

[90] Siemens. *Veloce Strato Emulation Platform.* https://eda.sw.siemens.com/en-US/ic/veloce/. 2021 (Accessed April 4th, 2021).

[91] SiFive. *HiFive Unleashed*. `https://www.sifive.com/boards/hifive-unleashed`. 2020 (Accessed November 25th, 2020).

[92] SiFive. *SiFive FU540-C000 Manual, v1p0*. `https://static.dev.sifive.com/FU540-C000-v1.0.pdf`. 2018.

[93] Synopsys. *Synopsys Unveils Next-Generation ZeBu Server 4*. `https://news.synopsys.com/2018-06-18-Synopsys-Unveils-Next-Generation-ZeBu-Server-4`. June 2018 (Accessed July 4th. 2020).

[94] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanović, and David Patterson. "DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs". In: *ASPLOS '15*. Istanbul, Turkey: ACM, 2015, pp. 207–221. ISBN: 978-1-4503-2835-7. DOI: `10.1145/2694344.2694362`. URL: `http://doi.acm.org/10.1145/2694344.2694362`.

[95] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. "RAMP Gold: An FPGA-Based Architecture Simulator for Multiprocessors". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 463–468. ISBN: 978-1-4503-0002-5. DOI: `10.1145/1837274.1837390`. URL: `http://doi.acm.org/10.1145/1837274.1837390`.

[96] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. "A Case for FAME: FPGA Architecture Model Execution". In: *Proceedings of the 37th annual international symposium on Computer architecture*. 2010, pp. 290–301.

[97] *Verilator*. `https://www.veripool.org/wiki/verilator`. (Accessed January 4th, 2021).

[98] Muralidaran Vijayaraghavan. "Theory of Composable Latency-Insensitive Refinements". MA thesis. Massachusetts Institute of Technology, June 2009.

[99] Muralidaran Vijayaraghavan and 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. Cambridge, Massachusetts: IEEE Press, 2009, pp. 171–180. ISBN: 978-1-4244-4806-7. URL: `http://dl.acm.org/citation.cfm?id=1715759.1715781`.

[100] Danyao Wang, Charles Lo, Jasmina Vasiljevic, Natalie Enright Jerger, and J Gregory Steffan. "DART: A Programmable Architecture for NoC Simulation on FPGAs". In: *IEEE Transactions on Computers* 63.3 (2014), pp. 664–678.

[101] Perry H Wang, Jamison D Collins, Christopher T Weaver, Blliappa Kuttanna, Shahram Salamian, Gautham N Chinya, Ethan Schuchman, Oliver Schilling, Thorsten Doil, Sebastian Steibl, et al. "Intel Atom Processor Core Made FPGA-Synthesizable". In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2009, pp. 209–218.

[102] Andrew Waterman. “Design of the RISC-V Instruction Set Architecture”. PhD thesis. EECS Department, University of California, Berkeley, Jan. 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html`.

[103] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C Hoe, Derek Chiou, and Krste Asanović. “RAMP: Research Accelerator for Multiple Processors”. In: *IEEE Micro* 27.2 (2007), pp. 46–57.

[104] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. “A Practical FPGA-Based Framework for Novel CMP Research”. In: *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. FPGA ’07. Monterey, California, USA: Association for Computing Machinery, 2007, pp. 116–125. ISBN: 9781595936004. DOI: `10.1145/1216919.1216936`. URL: `https://doi.org/10.1145/1216919.1216936`.

[105] Henry Wong, Vaughn Betz, and Jonathan Rose. “Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.10 (Oct. 2014), pp. 2067–2080. ISSN: 1063-8210. DOI: `10.1109/TVLSI.2013.2284281`.

[106] Kun-Cheng Wu and Yu-Wen Tsai. “Structured ASIC, Evolution or Revolution?” In: *Proceedings of the 2004 International Symposium on Physical Design*. ISPD ’04. Phoenix, Arizona, USA: Association for Computing Machinery, 2004, pp. 103–106. ISBN: 1581138172. DOI: `10.1145/981066.981088`. URL: `https://doi.acm.org/10.1145/981066.981088`.

[107] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling”. In: *SIGARCH Comput. Archit. News* 31.2 (May 2003), pp. 84–97. ISSN: 0163-5964. DOI: `10.1145/871656.859629`. URL: `http://doi.acm.org/10.1145/871656.859629`.

[108] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine”. In: (May 2020).