A Highly Productive Implementation of an Out-of-Order Processor Generator

Christopher Celio

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-151
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html

December 1, 2018

A Highly Productive Implementation of an Out-of-Order Processor Generator

by

Christopher Patrick Celio

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Emeritus David A. Patterson, Chair
Professor Krste Asanović, Co-chair
Assistant Professor Zachary Pardos

Fall 2017

A Highly Productive Implementation of an Out-of-Order Processor Generator

Abstract

A Highly Productive Implementation of an Out-of-Order Processor Generator

by

Christopher Patrick Celio

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Emeritus David A. Patterson, Chair

Professor Krste Asanović, Co-chair

General-purpose serial-thread performance gains have become more difficult for industry to realize due to the slowing down of process improvements. In this new regime of poor process scaling, continued performance improvement relies on a number of small-scale micro-architectural enhancements. However, software simulator-based models, which computer architecture research has largely relied upon, may not be well-suited for evaluating ideas at the necessary fidelity.

To facilitate architecture research during this fallow period of Moore's Law, we propose using processor simulators built from synthesizable processor designs. This thesis describes the design of a synthesizable, industry-competitive processor built on recent advancements in open-source hardware: we leverage the new open-source RISC-V instruction set architecture, the new *Chisel* hardware construction language, and the *Rocket-chip* processor generator.

Our processor generator is called BOOM, and it designed for use in education, research, and industry. Like most contemporary high-performance cores, BOOM is superscalar (able to execute multiple instructions per cycle) and out-of-order (able to execute instructions as their dependencies are resolved and not restricted to their program order).

The BOOM generator was implemented using the *Chisel* hardware construction language, allowing for the rapid implementation of parameterized designs. The *Chisel* description generates synthesizable implementations of BOOM that can target both FPGAs and ASIC tool-flows. The BOOM effort culminated in a test chip that was fabricated in the TSMC 28 nm HPM process (high performance mobile) using the foundry-provided standard-cell library and memory compiler.

This thesis highlights two aspects of the BOOM design: its industry-competitive branch prediction and its configurable execution datapath. The remainder of the thesis discusses the BOOM tape-out, which was performed by two graduate students and demonstrated the ability to quickly adapt the design to the physical design issues that arose.

To my parents,
for the many opportunities that they gave me.

Contents

List of Figures

List of Tables

Acknowledgments

A thesis is an enormous task, but thankfully, it is not something that is done alone. Many friends, family, and mentors contributed to this work in both tangible and intangible ways.

First, I would like to thank my advisors, Krste Asanović and Dave Patterson, for allowing me to pursue a topic that excited me, no matter how foolish I was being. Krste was always eager to dive into the details (like designing the freelist restore scheme), while Dave helped me focus on the big picture. I would also like to thank my other committee member, Zachary Pardos, for his guidance and Johnathan Bachrach for his participation in my qualification exam. I also appreciated Jonathan's foundational work on *Chisel* and his sincere enthusiasm for my project.

I owe my early interest in computer architecture to Chris Terman and to Anant Agarwal of MIT. Chris's excellent 6.004 lectures sparked my fascination with processors, and I am very thankful for his mentoring of my 6.111 project where I built my first FPGA processor. I have been hooked ever since. As a senior without a clear plan for the future, Anant took me into his lab and guided me to pursue graduate work in computer architecture.

Going back even further, I would like to thank Jeffrey Friedman of Rockefeller University for taking a crazy chance on a high school student. I am forever grateful for the patient mentoring by researchers in his molecular genetics lab, and in particular Esra Asilmaz and Gulya Fayzikhodjaeva. From them I learned that there are many important problems science can solve, and I wanted to be a part of that. I would also like to thank Geoff Crowley of Southwest Research Institute for his mentorship that helped further guide my path into research.

I would like to thank the ASPIRE Lab (and before that, the Par Lab) for providing a comfortable home for me and my work throughout grad school. Our corner cubicle, the *Cosmic Cube* (Scott Beamer, Sarah Bird, Henry Cook, and Andrew Waterman), helped make me feel at home, especially during my early years. I'd also like to thank Andrew for always answering my constant stream of questions, while he was busy building the stuff my research depended on.

I would like to give a big thank you to Donggyu Kim for bravely using BOOM as part of his own research, for collecting data, and finding many of the harder-to-find bugs. I would also like to thank Pi-Feng Chiu for her absolutely heroic effort in bringing BOOM to life in silicon form, and Professor Borivoje Nikolić for championing the tape-out.

Thanks to everyone who has sacrificed their time and sanity making *Chisel* a great language that is now used by real companies shipping real products. Many people have contributed, including Jonathan Bachrach, Huy Vo, Andrew Waterman, Jim Lawson, Chick Markley, Richard Lin, Jack Koenig, Adam Izraelevitz, Albert Magyar, Stephen Twigg, and Donggyu Kim.

Thanks to everyone who developed and improved *Rocket-chip*, and whose tape-out experiences helped Pi-Feng and I start our own tape-out standing on second base. Many people have contributed — the upstream open-source *Rocket-chip* has over 50 contributors and it is now used in many RISC-V companies' products — but in particular I would like to thank

Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman.

The ASPIRE Lab and Par Lab were unbelievably lucky in the quality and dedication of their staff, including Roxana Infante, Tamille Chouteau, Ria Briggs, Kostadin Ilov, and Jon Kuroda. In particular, I would like to thank system administrator Kostadin for working miracles and doing everything in his power to help every student. Even when all of Lake Tahoe got snowed out for the Winter Retreat in 2017, Kostadin still managed to make it there and set everything up anyways.

Speaking of which, I would also like to thank all of the industry sponsors and attendees of our lab's biannual retreats. It is one of the things that has made my graduate school experience unique, and I appreciate the enthusiasm and support provided to me and my crazy venture. Specifically, I would like to thank Mark Rowland for his advice, support, and entertaining war stories that helped me feel that what I was doing mattered.

Triple Rock, Jupiters, and Beta Lounge also deserve recognition for providing a constant and comfortable collaboration space in which to grow friendships and share ideas over the years. Often I found that I was too busy during the working hours to get any work done, and it was only afterwards when the real research discussions could commence, away from computers and email.

I would like to thank my best friend and wife Kimberlee, for her support throughout grad school (and especially while I lingered in the finishing up stage), and for being a sounding board for many of my ideas. You have been my biggest cheerleader, and that has meant the world to me.

Finally, to my family. My brother Andrew Celio, for being my first and longest friend. And to my parents, for guiding me, looking out for me, and always being engaged. I appreciate that I got my first taste of "research" building all manner of contraptions in our garage together.

Chapter 1

Introduction

Computer architecture research centers on the study and evaluation of computer systems, their organization, their interfaces, and their implementations. And yet, due to the complexity of modern computer processors, architecture researchers only rarely design and implement entire systems.

There are many valid reasons to eschew the arduous process of implementing full systems. The man-power and the time required to take a project from conception to completion is high. The level of re-use between chip-building research projects is also poor, exacerbating the amount of effort required by each new team to construct a complete system. And certainly, many ideas can be adequately evaluated using software simulators, where researchers can add their one sliver of innovation to an existing research platform.

But some ideas are best evaluated in the context of a full system, and with enough detail to provide performance, power, and area numbers. This thesis aims to provide a piece of the puzzle by implementing — at the register-transfer-level (RTL) — a complete out-of-order processor. We focus not just on the design, but on the manner in which it has been productively produced using a new, more modern hardware construction language. We also discuss how we implemented not just a single processor instance, but rather, a hardware generator that can construct an entire family of out-of-order cores. We call our processor generator the *Berkeley Out-of-Order Machine*, or BOOM.

1.1 Leveraging New Infrastructure

The feasibility of BOOM is in large part due to the available open-source infrastructure that has been developed in parallel at UC Berkeley.

BOOM implements the open-source RISC-V Instruction Set Architecture (ISA), which was designed from the ground-up to enable VLSI-driven computer architecture research. The RISC-V ISA is clean, realistic, and highly extensible. Available software includes the *GCC* and *LLVM* compilers and a port of the Linux operating system. The clean and simple design of RISC-V allowed us to focus on our processor generator's design without getting weighed

down with awkward instructions that demand undue attention. And the advantage of an open-source community-driven ISA allowed us to leverage an existing tool-chain and software base without spending time away from processor implementation to manage software porting efforts.

BOOM is written in *Chisel*, an open-source hardware construction language developed to enable advanced hardware design. *Chisel* allows designers to utilize concepts such as object orientation, functional programming, parameterized types, and type inference which makes it easier to implement highly parameterized hardware generators. From a single *Chisel* source, *Chisel* can generate a cycle-exact Verilog software simulator, Verilog targeting FPGA designs, and Verilog targeting ASIC tool-flows. One of *Chisel*'s strengths is its focus on generating well-formed, synthesizable Verilog. This feature decreased design risk. *Chisel* also brings software development-level productivity to RTL coding. Other hardware description languages feel unwieldy with many common design patterns being awkward or verbose to describe. But *Chisel* lets us focus more time on the ideas we wanted to express, and less time on figuring out *how* to express them.

UC Berkeley also provides the open-source *Rocket-chip* System-on-a-Chip (SoC) generator, which has been successfully taped-out over a dozen times in multiple different, modern technologies by multiple groups [4, 57, 107]. BOOM makes significant use of *Rocket-chip* as a library – the caches, the uncore, and the functional units all derive from *Rocket*. In total, over 11,500 lines of code is instantiated by BOOM from the *Rocket-chip* repository.

1.2 Contributions

This thesis makes the following contributions:

- **A complete implementation of a superscalar, out-of-order processor generator** — We built a superscalar, out-of-order processor generator. BOOM implements the entire RISC-V RV64G ISA and the page-based virtual memory Sv39 Privileged ISA such that we can boot the Linux operating system and run user-level applications.
- **A competitive implementation** — BOOM achieves comparable (or better) branch prediction accuracy and instructions-per-cycle performance relative to similarly sized industry out-of-order processors.
- **A productive implementation** — We demonstrated our productive processor generator design by implementing it using only 16k lines of code. We were able to accomplish this task by leveraging many new artifacts in the nascent open-source hardware ecosystem. Specifically, we were able to leverage the open-source RISC-V ISA and its accompanying tool-chain and testing infrastructure, the open-source *Chisel* hardware construction language, and the open-source *Rocket-chip* SoC generator.

- **Demonstrated productivity with an Agile tape-out** — We further demonstrated our productivity and agility by making significant micro-architectural design changes as part of a two-person tape-out performed over four months.

1.3 Thesis Outline

In Chapter 2 we discuss the state of the industry and of research, and of the technology trends that are motivating changes in both. In particular, there is a growing requirement for high fidelity RTL-based simulations that can provide greater system depth and a higher degree of confidence in performance and power estimations. In Chapter 3 we provide an overview of our superscalar, out-of-order processor generator called BOOM. Chapter 4 discusses in detail our implementation of the branch prediction and instruction fetch pipelines, and how we were able to implement complex branch predictors that can be easily modified and changed to explore new ideas. In Chapter 5 we discuss in more detail how we leveraged the *Chisel* language to productively describe a superscalar, out-of-order datapath generator; we also discuss how we leveraged expert-written functional units to increase our productivity in describing a full processor system capable of executing floating-point applications. And finally, in Chapter 6 we demonstrate our productivity through a case study in which we taped-out an instantiation of BOOM as part of a test-chip fabricated using the TSMC 28 nm HPM process.

Chapter 2

Background

This chapter provides the background and motivation for implementing the open-source *Berkeley Out-of-Order Machine* (BOOM). Section 2.1 discusses the importance of computers and provides motivation for studying computer architecture. Section 2.2 discusses the trends in technology regarding Moore's Law and Dennard Scaling and their effect on processor design and computer architecture research. Section 2.3 discusses the current state-of-the-art in computer architecture research regarding processor simulators as well as related work regarding similar register-transfer-level (RTL) implementation efforts. Section 2.4 describes the organization, or *microarchitecture*, of a typical, modern processor and introduces the *out-of-order* microarchitecture. Section 2.5 provides a taxonomy of out-of-order processors while Section 2.6 discusses the history of out-of-order processors. Section 2.7 elaborates on the performance advantages that out-of-order processors have over other processor designs and Section 2.8 concludes.

2.1 Motivation

For the past half-century, computers have been a key enabler in technological and societal change. The number of processors worldwide has continued to proliferate with little sign of slowing despite the industry upheaval caused by the current slowing of Moore's Law and the end of Dennard Scaling in 2005.

Computer usage has been growing on many fronts. For cell phones, the numbers are absolutely staggering. From the United Nation's World Telecommunication/ICT Indicators Database, over 6 billion people own a cell phone [116] (see Figure 2.1). As a point of comparison, only 4.5 billion people have access to toilets or latrines [115]. For many, cell phones have been an enabler for economic development and societal progress [1, 28, 96].

Cellphones are not the only benefactors of computer architectural innovation. The growth of *Software as a Service* (SaaS) has led to a rise of a new class of computer – the Warehouse-Scale Computer (WSC). There are many demanding applications running on WSCs: search, media delivery, gaming, shopping, social networking, and machine learning. WSCs are typi-

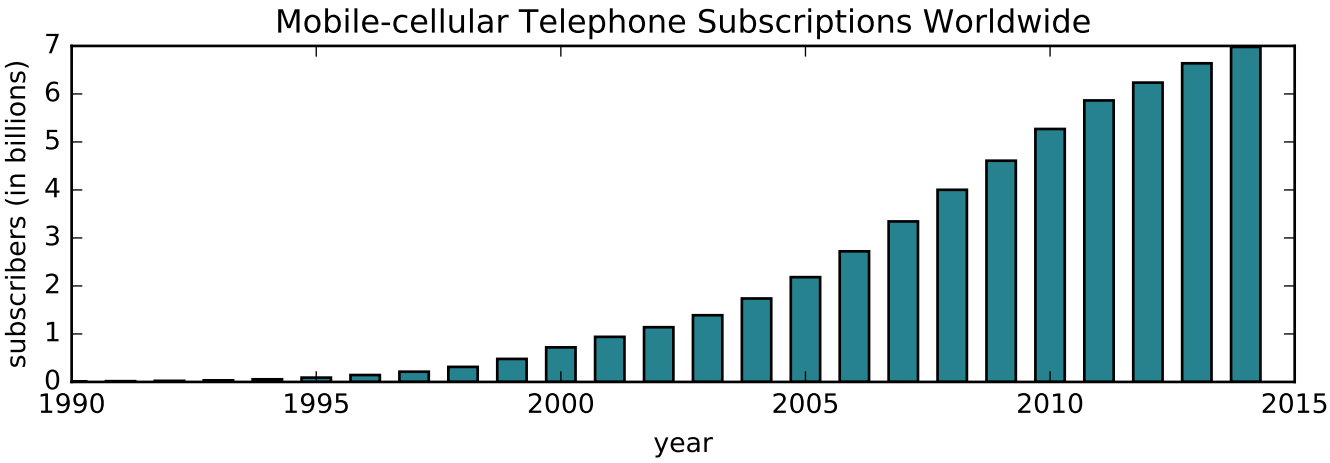


Figure 2.1: Cell phone subscriber counts as provided by the United Nations [116].

cally built in locations where the price of power is cheap as they require megawatts to power (and cool) tens of thousands of compute nodes [77]. Over a ten year period, 30% system cost of a WSC may go to energy, power distribution, and thermal management [77]. However, using wimpier but more power-efficiency cores can dramatically degrade service response times, making the trade-off between performance and power-efficiency a non-trivial design problem [45].

At the opposite end of the compute spectrum, the *Internet of Things* (IoT) promises to create a world of billions of "smart" sensors, dwarfing the current number of deployed processors [9]. However, each device will likely have to be less than $1 and able to sustain itself on a tiny power budget – perhaps sipping the necessary power from the environment itself [80].

While the number of processors will continue to grow into the near future at all parts of the compute spectrum, from small microwatt processors (IoT) to megawatt computers (WSC), there are a number of challenges that designers face such as cost, reliability, service response time, thermal constraints, and energy-efficiency [45].

2.2 Technology and Research Trends

Changes in semiconductor technology have provided computer architects with many new tools with which to pursue higher performance processors in a variety of computing environments. *Moore's Law* has provided architects with more (and faster) transistors with which to implement more complex designs while *Dennard Scaling* has allowed the power requirements to stay manageable. However, the end of Dennard Scaling has made power a first-order design constraint and seriously limited full-frequency transistor utilization.

In 1965, Gordon Moore released a report discussing the trends in transistor technology.

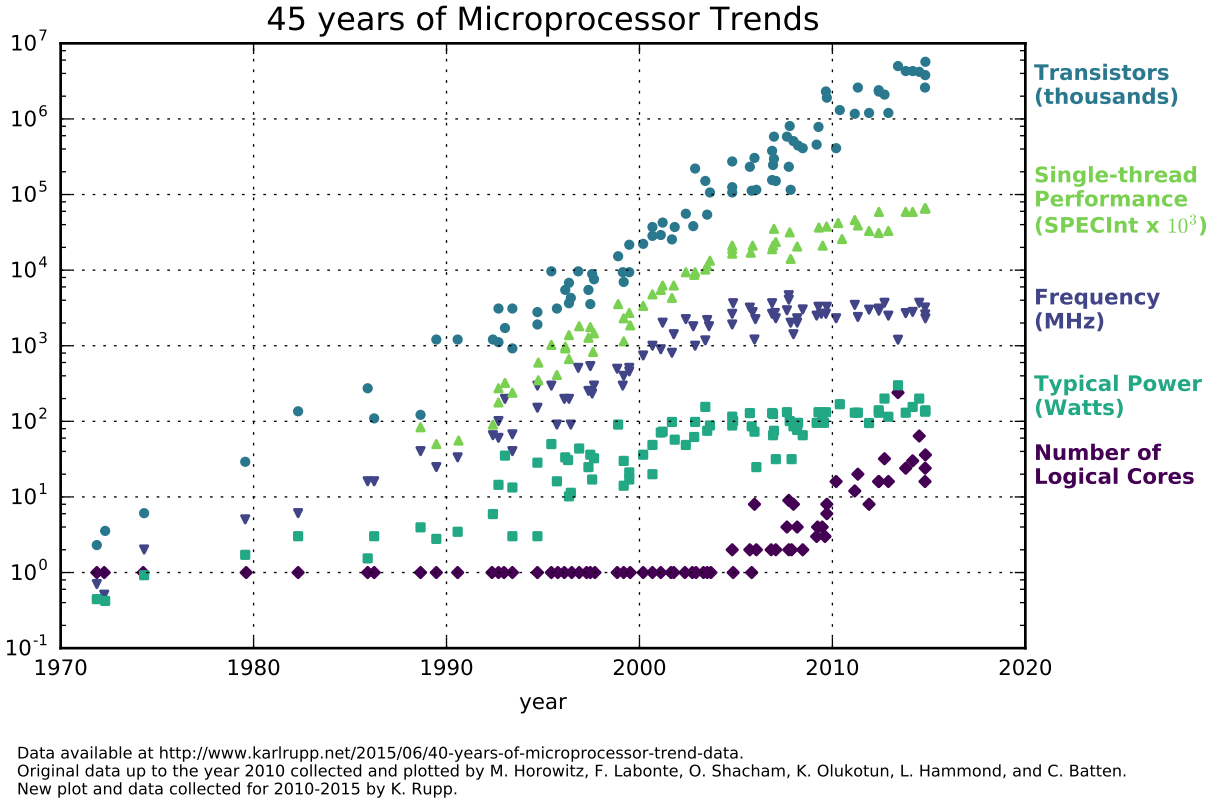


Figure 2.2: Although signs of slowing have begun to show, transistor counts have otherwise continued to scale as predicted by Moore's Law for over 45 years. However, around 2005 processors began to be restricted by power and thermal budgets at around 100 watts (the "power wall") as Dennard Scaling came to an end. This new power constraint prevented frequency from scaling any further, which in turn adversely affected the scaling of single-core performance. Unable to improve scalar performance at the same rates as before, transistor budgets have been redirected to focus on other techniques for delivering value to the user. One technique has been to focus on increasing parallel performance by increasing the number of logical cores.

He noted that electronics became cheaper as more transistors could be placed onto a single circuit chip reducing the total number of chips needed. However, as the number of transistors increased, so did the probability of having a bad transistor, lowering the manufacturing yield and increasing the cost of each chip. These two economic forces worked against each other. But as manufacturing yield improves, cost-efficiency motivates smaller and smaller transistors. In short, transistors would continue to become smaller very quickly, incentivized by economics [72].

Moore noticed that the number of transistors per chip was doubling every year and — from only five data points — predicted that this trend would continue for "at least 10 years" into the future. Indeed, "Moore's Law" — that the most cost-effective transistor size would continue to shrink leading to a doubling of transistor counts roughly every 18 to 24 months — has been maintained for over 50 years.

Three years after publishing his report, Gordon Moore, along with Robert Noyce, founded Intel (originally called NM Electronics). Their first microprocessor, the Intel 4004 used 2,300 transistors in 10 um technology in 1971. 45 years later, the Intel Broadwell-EP Xeon E5-2600 V4 shipped with 7.2 billion transistors using 14 nm technology [79]. Although the number of foundries who can afford the rising costs of pursuing smaller technology nodes has dwindled, some foundries have nonetheless released details of their intentions for delivering 7 nm technology [51].

Moore's Law has provided computer architects an abundance of transistors with which to implement new ideas and techniques for improving processor performance. However, while Moore's Law may end soon [70], the end of Denard Scaling ten years ago signaled the beginning of a real crisis in the industry.

Table 2.1 shows the transistor scaling factors under a Dennard Scaling regime. By scaling voltage, doping, and transistor size together, the signal delay through the transistor could be increased by the same factor while maintaining the same power density as before. In summary, under Dennard Scaling, as a transistor's length and width were halved, new processors could have four times as many transistors, be clocked twice as fast, and use the same amount as power as the previous generation. This led to a golden era of computer architecture in the 1980s and 1990s as transistor counts doubled every 18 months and processor clock frequencies doubled every 36 months. Software became faster by the virtue of Dennard Scaling, and computer architects could explore more complex and transistor-hungry designs to exploit the growing transistor budgets while still hitting their power budgets. Technology improvements working in concert with microarchitectural innovations has led to a 50,000x increase in single-core processor integer performance over the last 45 years as shown in Figure 2.3.

Although the scaling had never been perfect, around 2005, technology scaling fully diverged from the ideal scaling rates described by Dennard, as the voltage could no longer be scaled even as transistors continued to become smaller [104]. As the gate oxide thickness (t_{ox}) grew smaller (which directly sets the threshold voltage t_{th}), the current leakage across the gate grew, leading to a rising *static* power and a reduction in transistor reliability [46]. This static power is now significant enough to be a considerable factor in a processor's power

Table 2.1: The scaling factors of a single transistor under a Dennard Scaling regime (1965-2005) [27]. By scaling the voltage and doping along with the size of the transistor each by κ, each transistor can now be clocked κ times faster while maintaining the same power density as the previous generation transistor.

Device or Circuit Parameter	Dennard Scaling Factor
Device dimension (t_{ox}, L, W)	$1/\kappa$
Device quantity (A)	κ^2
Doping concentration (N_a)	κ
Voltage (V)	$1/\kappa$
Current (I)	$1/\kappa$
Capacitance ($\epsilon A/t$)	$1/\kappa$
Delay time / circuit (VC/I)	$1/\kappa$
Power dissipation / circuit (VI)	$1/\kappa^2$
Power density (VI/A)	1
Utilization (1/Power)	1

Table 2.2: The scaling factors of a single transistor in a post-Dennard Scaling regime (post-2005). As voltage can no longer be scaled down with transistors, each transistor now dissipated the same power as before leading to an increase in power density equal to the increase in transistor density. Constrained by power and thermal requirements, this leads to successive generations of processors that are unable to use more and more of its transistors at full frequency at a rate of $1/\kappa^2$. This inability to utilize all of the chips transistors at full frequency simultaneously is known as *Dark Silicon* [104].

Device or Circuit Parameter	Post-Dennard Scaling Factor
Device dimension (t_{ox}, L, W)	$1/\kappa$
Device quantity (A)	κ^2
Voltage (V)	1
Capacitance ($\epsilon A/t$)	$1/\kappa$
Delay time / circuit (VC/I)	$1/\kappa$
Power dissipation / circuit (VI)	$1/\kappa^2$
Power density (VI/A=AFC^2)	κ^2
Utilization (1/Power)	$1/\kappa^2$

budget. With the threshold voltage unable to scale any further, the supply voltage (V_{dd}) scaling has also stopped. The supply voltage V_{dd} provides a quadratic relationship to the processor's *dynamic* power [24].

$$Power_{cpu} = Power_{static} + Power_{dynamic} \tag{2.1}$$

Static power is largely a function of the leakage current across the gate which worsens as the threshold voltage is lowered.

Dynamic power is dissipated on every transistor state transition, and is thus dependent on how often and how many transistors flip from off to on (and vice versa). The α term denotes what fraction of transistors transit every cycle and is dependent on the workload, f denotes the clock frequency, and C is the *load capacitance* of the processor.[1,2]:

$$Power_{dynamic} = \alpha * C * V_{dd}^2 * f \tag{2.2}$$

As frequency f is typically coupled to V_{dd}, this provides the following relationship to dynamic power:

$$Power_{dynamic} \propto V_{dd}^3 \tag{2.3}$$

With the supply voltage staying constant in a post-Dennard Scaling regime, each transistor now dissipates the same amount of power as before, leading to growing power density as the number of transistors on a chip continues to increase. As Moore's Law has continued past the end of Dennard Scaling, this has meant that processors can no longer switch every transistor at full frequency without exceeding the power or thermal limitations of the chip. Instead, computer architects have had to rely on a number of techniques to manage the new "power wall" limitations [104]. Table 2.2 describes this new scaling regime.

One change was a move to multicore processors. Instead of a single, monolithic, high frequency core, processors moved to a modest number of processor cores that were clocked at a slightly slower frequency than the previous generation. As shown by Equations 2.2 and 2.3, a slight reduction in frequency allows for a slight reduction in voltage which leads to a significant reduction in power. However, the move to multicore processors has been a significant burden on programmers. One problem with the shift to multicore processors is the fact that most applications have a fundamental serialization limit, codified by *Amdahl's Law*. This serialization bottleneck restricts the maximum potential speedup from parallelizing a particular algorithm [2].

Another approach to managing the power wall has been to rely on specialization. A particular algorithm may only utilize a small, but highly specialized fraction of the chip. As the program goes through different phases and utilizes different algorithms for each phase,

[1]These equations assume the voltage swing is equal to the supply voltage [99].

[2]Particularly for high-frequency processors, there is an added "short-circuit" power dissipation term that occurs when both transistors in a CMOS gate are on simultaneously, connecting the power source to ground, and is a function of activity factor α and frequency f [109].

different parts can be turned on or off as needed. Specialized circuits, known as *accelerators*, have been used for cryptography, compression, graphics, and machine learning. Accelerators can provide performance and energy-efficiency improvements in the range of 100-1000x over a general-purpose processor core [104].

2.2.1 Impact of Performance Slowdown on Architecture Research

The end of Dennard Scaling has led to significant challenges in the field of computer architecture. Processor frequency gains and single-core performance has stalled. As shown in Figure 2.3, it is becoming more and more challenging to realize additional gains in single-core performance. For this reason, we believe it is becoming more important to simulate and evaluate microarchitectural ideas at a higher level of fidelity than was needed in the past.

There are many different levels of fidelity for describing a processor. High-level, *cycle-approximate* models are used to simulate a processor to provide a high-level view of how the performance might change.

The register-transfer-level (RTL) is a level in which the processor description encompasses the behavior and movement of data between hardware registers. Although RTL was originally used as a relatively high-level simulation abstraction to provide a cycle-accurate simulation of a design, VLSI tools have since evolved to map RTL descriptions directly to physical realizations.

While high-level, cycle-approximate models provide the flexibility to quickly iterate and explore the design space, they have significant blind spots. RTL models provide a list of advantages. First, coupled with the appropriate CAD tools, RTL implementations can provide area, power, and timing information that is not available in higher-level models. Second, RTL implementations are more likely to be grounded in reality. This grounding has a number of advantages. It provides a higher degree in confidence, it can prevent correctness errors from changing research conclusions, and it can motivate new design ideas that must manage realistic constraints.

Despite industry's best efforts, the gains in single-thread performance has continued to slow. From the data from [53] and [78] (Figures 2.2 and 2.3), the SPECINT performance has increased by only around 9% per year in the last six years. The most recent Intel processor, the 2017 Kaby Lake, uses the same technology process and the same microarchitecture as its 2015 predecessor Skylake. Kaby Lake represents as "optimization" stepping in which Intel utilized low-level optimizations to provide a 7% improvement in the peak clock frequency. However, this peak frequency can only be realized by a small fraction of the processor at any one point in time. Intel calls this "Turbo Boosting" in which a single core may be clocked at a higher frequency while keeping the other cores in a lower energy state so long as the total processor's power and thermal requirements are maintained.

To realize these continued gains often requires a few new microarchitectural tricks, each providing only a few percent gains each in performance. Thus, to evaluate a new, industry-

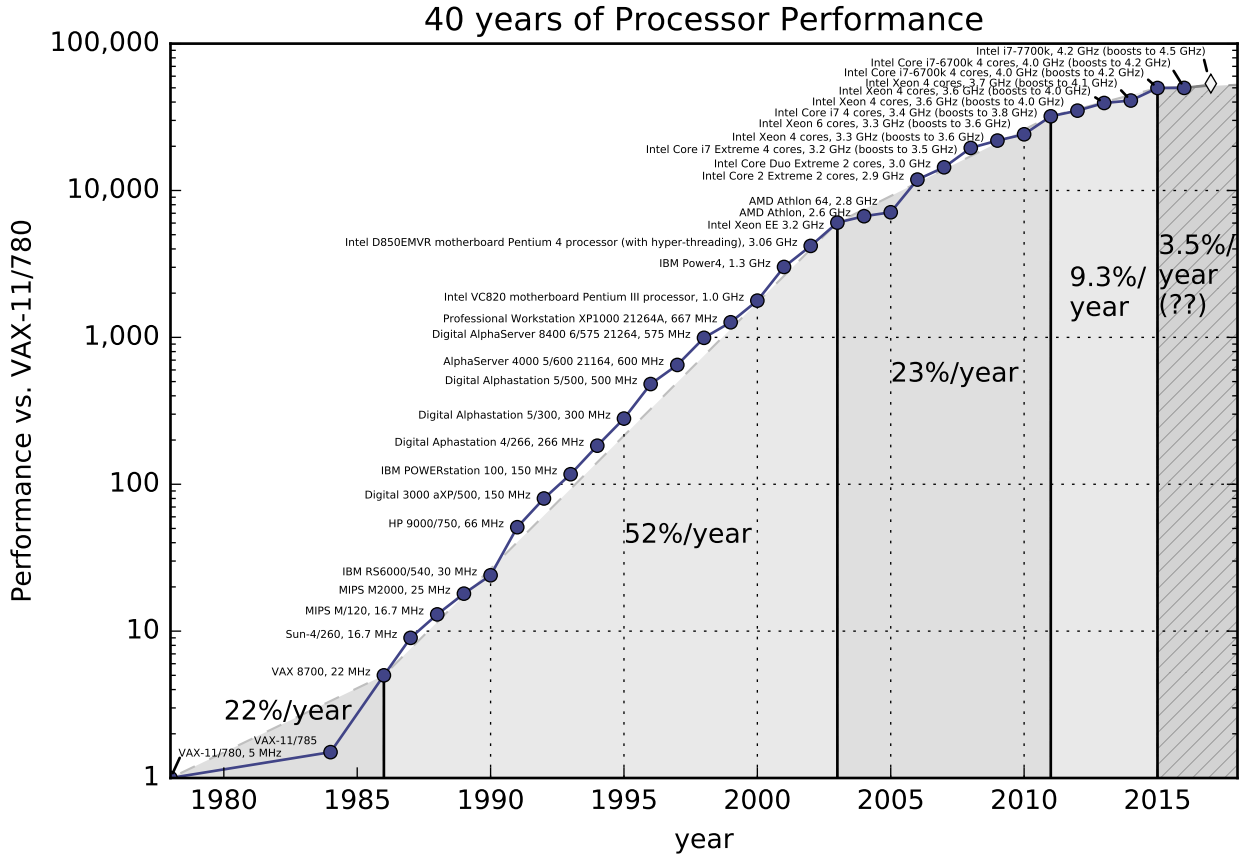


Figure 2.3: Microarchitectural ideas for improving performance are becoming harder and harder to come by. Data is shown comparing SPECIntbase performance relative to a VAX-11/780. Early performance gains were largely technology driven. In the 1980s and 1990s, computer architectural innovations – such as the ability to recognize and exploit instruction-level parallelism (ILP) – worked together with continued technological improvements to deliver even greater performance gains. However, the end of Dennard Scaling in 2005 caused processor frequencies to stall. Around the same time, the remaining ILP had become more difficult to extract. Both of these factors have conspired to slow single-core performance gains. Performance once again appears to be limited to technology improvements and circuit-level techniques that provide more efficient usage of power. As SPEC is updated over time, performance is estimated by use of a scaling factor to normalize across different versions of SPECInt (SPEC89, SPEC92, SPEC95, SPEC2000, SPEC2006). The scaling factor is empirically determined by running multiple versions of SPEC on a subset of the provided machines. The SPEC performance of the just-released Kaby Lake i7-7700k is synthetically created by comparing the relative Geekbench results [37] and the clock frequency improvement to the Skylake i7-6700k, with which it shares the same micro-architecture and process technology. Data is provided by [78].

worthy idea requires a significant degree of fidelity to provide the necessary confidence. Current methodologies are geared towards finding integer factor improvements, and are not well suited for exploring small wins on the order of 2-3%. These small gains suggests a requirement for higher fidelity methodologies that include utilizing RTL implementations.

2.3 Computer Architecture Research Trends

In a perfect world, computer architecture researchers would design, implement, and then fabricate their research designs to fully analyze and evaluate their ideas. Although there are a few examples of such ambitious research projects [88, 105, 57, 100], this methodology is largely infeasible for a number of reasons. First, the feedback loop from generating an idea to measuring a single implementation takes many years; the man-power and fabrication costs can be prohibitive; and some research ideas attempt to explore future designs that speculate technological abilities that are not yet available.

For these reasons, most computer architecture research relies on simulators to model processors. Simulators can allow a very small team of researchers to quickly implement and explore a design space before providing recommendations. Even industry first relies on insights gained by simulators before spending resources implementing a physical design.

Ideal simulators attempt to achieve the following goals:

1. The **fidelity** of the simulator – does it accurately model the intended target?
2. The simulation **speed** of the simulator – how many design points (or workloads) can we explore in a fixed period of time?
3. The **scope** of the simulator – does it provide area, power, and timing insight into the design?
4. The **flexibility** of the simulator – can new ideas be quickly and easily implemented and evaluated?

Unfortunately, these traits are in competition; there is no perfect simulator.

2.3.1 Architectural Simulators

Computer architecture research is largely carried out using simulators to model future hypothetical processor designs. Unfortunately, most research has focused on ease of implementing new ideas at the cost of fidelity, speed, and accuracy. Attempts to improve these metrics have largely be zero-sum with the ease of implementing new ideas.

SimpleScalar, released in 1996, models an out-of-order processor which implements a simplified derivative of the MIPS IV ISA [17]. However, SimpleScalar can only execute user-level code making it difficult to explore interactions between user and privilege-level code

like operating systems [16]. Users would also have to compile their benchmarks specifically for SimpleScalar and could not study commercial or off-the-shelf software.

Simics is an functional architectural simulator released in 2002. It executes unmodified software programs and is able to simulate a full system including supervisor instructions and provides interfaces to many different I/O devices. However, it only provides functional simulation and provides no microarchitectural or timing information [61]. Later efforts from other groups have added their own timing models on top of Simics [62].

The SESC ("Super ESCalar") simulator (2004) splits the execution model from the timing model. One advantage of this separation of concerns is it allows small errors in the timing model to not affect the correctness of the simulation. In a few cases, SESC developers purposefully eschew rare corner-cases that could affect the simulation speed or complexity of the simulator. SESC is not a full-system simulator and must trap all application system calls and emulate them [75]. Although it targets a MIPS ISA, it requires a custom toolchain to build applications to run on SESC. SESC is around 120k lines of code spread across nearly 400 files [52].

The QEMU emulator (2005) is a very fast functional full-system emulator. It uses dynamic binary translation to translate instructions from the target architecture into a sequence of host instructions. QEMU then caches, or remembers, these translations for future iterations through the program. This optimization allows QEMU to run close to native host processor speeds, simulating over 1 billion instructions per second (1 GIPs) [12]. QEMU, by itself, provides no timing modeling.

The gem5 simulation infrastructure (2011) provides both microarchitectural timings models and a full-system simulator able to simulate many different commercial ISAs. The gem5 system provides a flexible, modular framework that can utilize a number of processor models, memory system models, and network models. The range of models of varying complexity and accuracy allow users to trade off simulation speed for fidelity. Gem5 also provides a full system simulation mode which simulates user-level and privilege-level software while also providing models of device drivers for the OS to interface with. Checkpoint saving and restoring support is available, allowing researchers to avoid painful start-up overheads by loading up the same architectural state from a particular workload onto different microarchitectures under study [14].

MARSS (Micro Architectural and System Simulator) is a simulator (2011) built on top of the QEMU emulator [12]. When executing in a low-level, cycle-approximate detailed simulation mode, it is able to simulate a multicore processor at a simulation speed of 200 to 400 kilo instructions per second (KIPs) [76].

An updated version of SESC, called ESESC ("Enhanced SESC"), also uses QEMU for its execution model, allowing it to simulate the ARM instruction set. ESESC approaches simulating multi-threaded applications across multiple cores using Time-Based Sampling (TBS). The ESESC developers found the average error of the time-based sampling method to be within 4.99% compared to full simulation for a number of target configurations and applications [5]. ESESC integrates with the McPAT energy modeling tool to provide models for power and temperature. By utilizing sampling, ESESC can achieve up to 9 MIPs. The

full ESESC repository contains over 14 million lines of C and C++ code, which includes the tool chains, the QEMU simulator, and the McPAT energy model. The ESESC-specific timing models are more manageable at 100k lines of code and the thermal and power code is 300k lines of code [32].

SimFlex is another simulator that relies on statistical sampling to provide wider coverage across long-running workloads. Flexus is a full-system simulation infrastructure built on top of Simics. SimFlex is a methodology built on top of Flexus which uses a mixture of restoring workload checkpoints, fast microarchitectural warming, and detailed simulation of small simpoints to approximate the performance of a full workload [121].

Simulating processor cores at a high level of fidelity naturally leads to very slow simulations — the simulation model (which runs on the order of 100s of kilohertz) is typically on the order of 10,000 times slower than a real processor (running in the range of a gigahertz). As processors move to using multiple cores, the simulation slow-down is exacerbated. Tan [103] reported that the number of simulated cycles per benchmark in ISCA in 1998 was the same in 2008, but that the simulated core counts had increased from 1 to 16, diluting the number of simulated cycles per core.

The Sniper Multi-core Simulator tackles this problem by simulating each core at a higher level of detail using "interval simulation" [38, 43]. Instead of simulating the core at a cycle-approximate level, Sniper builds an analytical model as it executes a program. It begins by breaking up programs into "intervals" based on the "miss events", which are expensive, slower operations such as cache misses or branch mispredictions. Sniper then analyzes how these expensive miss events interact and provides an estimation for how much time each interval adds to the simulation time. This higher level of abstraction allows Sniper to run at "around 1 MIPS for an 8-core simulation" [5]. Sniper was validated against multi-socket Intel Core2 and Nehalem systems and provides average performance prediction errors within 25% while achieving a speed of up to several MIPS [43].

Simulating even higher core counts has proven difficult, as the additional cores further degrade the simulation speed. Ideally, each simulated ("target") core could be simulated on a single "host" core. However, to provide a cycle-accurate model, the host cores would have to synchronize after simulating every target cycle. Also, any communication between the target cores would have to be handled between the host cores before the next cycle of target simulation could begin. The goal of Graphite is to enable thousand-core processor simulations. Graphite accomplishes this by being a distributed parallel multi-core simulator that relies on relaxing the strict ordering of events between the simulated cores. Each target core is broken into two pieces — a core model and a memory model — each piece is simulated via its own host thread. The threads are then scheduled across a cluster. When target cores send data to one another, they also synchronize their clocks, allowing core clocks to skew within a bounded amount of time. Running 1024 cores on eight 8-core processors exhibited a 41x slow-down from native execution [69]. Graphite is built on top of Pin to provide dynamic binary translation [59]. Pin allows off-the-shelf x86 ISA applications to be functionally simulated while also providing event handlers to hook into the Graphite timing model.

ZSim is another thousand-core simulator that utilizes dynamic binary translation techniques to create "instruction-driven timing models", as opposed to "event-driven models". It also introduces a technique called "bound-weave", a two-phase technique in which cores may skew events with respect to each other for a bounded period of time for the first phase, and in the second phase replay the trace of events to determine the actual latencies [85].

A major downside to all of these software simulators is the roughly ≈10,000x simulation penalty to model any level of microarchitectural timing. One solution is to run the simulator on a Field Programmable Gate Array (FPGA). FPGAs are silicon chips that are "programmable" – RTL hardware designs may be synthesized and "simulated" on the FPGA. As FPGAs run on the order of 10s to 100s of MHz, FPGA-based simulators can provide orders of magnitude increases in simulation speeds [23, 102].

Ramp Gold is a one example of an FPGA-based simulator [103, 102]. It simulates 64 SPARC in-order cores on a single $750 FPGA board. It accomplishes this by a) decoupling the FPGA's clock from the simulated target clock (many FPGA clock cycles are required to simulate a single target cycle), b) abstracting some of the uncore timing models to allow for higher-level descriptions, and c) multi-threading the single, physical processor pipeline and sharing it across 64 simulated target cores. RAMP Gold's FPGA clock ran at 50 MHz, or 0.0156 MHz target frequency, for a speedup of roughly $250x$ relative to a detailed Simics+GEMS simulation of a similar target platform. The Ramp Gold functional model is roughly 35,000 lines of SystemVerilog while the timing model is only 1,000 lines of SystemVerilog [103]. However, Ramp Gold's major downside is the difficulty in modifying Ramp Gold to explore other processor design points. The timing model and the functional model are fairly wedded to simulating homogenous, in-order cores.

Many of the simulators discussed all suffer from similar problems: 1) many do not support full systems, 2) they suffer from slow simulation speeds (generally less than 200 kHz for software simulators [85]), 3) they cannot produce area, power, or timing numbers, and 4) they produce hard to trust or verify results when exploring new designs that look significantly different from current industry designs.

Unfortunately, the challenges imposed by the power constraints brought on by the end of Dennard Scaling make it vitally important that microarchitectural ideas are evaluated not just by their effect on performance but also on area and power. Perhaps more importantly, the significant slow-down in single-core performance improvements necessitate that any ideas we explore must be done so with high fidelity.

2.3.2 RTL Implementations

Due to the limitations of simulators as described in the previous section, there have been a few academic efforts to implement cores at the register-transfer-level (RTL). We will focus on out-of-order cores which attempt to capture the highest level of general-purpose single-core performance.

The Illinois Verilog Model (IVM) is a "latch-accurate" model of a 4-issue, out-of-order core designed to study transient faults [119]. Written in Verilog-95, IVM originally relied

Table 2.3: A sample of simulators. Many modern simulators use dynamic binary translation (DBT) to accelerate execution, typically leveraging existing functional emulators QEMU [12] or Pin [59]. Data compiled with help from [85].

Simulator	year	Engine	Parallelization	Detailed Uarch	Full System	unmodified binaries
SimpleScalar [16]	1996	emulation	sequential	OOO	no	no
Simics [61]	2002	emulation	sequential	no	yes	yes
SESC [75]	2004	emulation	sequential	OOO	no	no
QEMU [12]	2005	DBT	sequential	no	yes	yes
Graphite [69]	2010	DBT (Pin)	skew	approx-IO	no	yes
Ramp Gold [103]	2010	FPGA	sequential	IO	no	yes
gem5 [14]	2011	emulation (m5)	sequential	OOO	yes	yes
MARSS [76]	2011	DBT (QEMU)	sequential	OOO	yes	yes
Sniper [43]	2012	DBT (Pin)	skew	approx-OOO	no	yes
ESESC [5]	2013	DBT (QEMU)	sampling	OOO	yes	yes
ZSim [85]	2013	DBT (Pin)	bound-weave	DBT-OOO	no	yes

Table 2.4: A sample of academic out-of-order processors and the open-source UltraSPARC T2. The UltraSPARC T1, also open-source, was modified by Princeton to create the OpenPiton many-core processor [11].

	IVM [119]	SCOORE [10]	FabScalar [29, 84]	Sharing [126]	BOOM	UltraSPARC T2 [74]
industry						√
fully synthesizable	√		√	√	√	√
FPGA		√	√		√	√
parameterized			√		√	
floating point		√			√	√
atomic support					√	√
L1 cache	√	√	√	√	√	√
L2 cache			√	√	√	√
virtual memory					√	√
boots Linux					√	√
multi-core			√		√	√
ISA	Alpha (sub-set)	SPARCv8	PISA (sub-set)†	Alpha (sub-set)	RISC-V	SPARCv9
lines of code	30,000	?	75,000†	31,900	9,000 + 11,500	1,900,000

†Information was gathered from publicly available code at [33].

on some unsynthesizable constructs (such as `while` loops that do not terminate on constant synthesis-time values), but it has since been patched to be fully synthesizable [24].

The Santa Cruz Out-of-Order RISC Engine (SCOORE) was designed to efficiently target both ASIC (1 Ghz at 90 nm) and FPGA generation (200 MHz) [67]. Unfortunately, SCOORE lacks a synthesizable fetch unit and was never completed.

FabScalar is a tool for composing synthesizable out-of-order cores. It searches through a library of parameterized components of varying pipeline width and depth to find an optimal core design for a given benchmark set, guided by performance constraints given by the designer [24]. FabScalar has been demonstrated on an FPGA [29], however, as FabScalar did not implement caches, all memory operations were treated as cache hits. Later work incorporated the OpenSPARC T2 caches in a tape-out of FabScalar [84].

The Sharing Architecture is composed of a two-wide out-of-order core (or "slice") that can be combined with other slices to form a single, larger out-of-order core. By implementing a slice in RTL, they were able to accurately demonstrate the area costs associated with reconfigurable, virtual cores [126].

Unfortunately, there are currently no open-source industry implementations of an out-of-order core. However, Sun has released two of their SPARC v9 server-level processors, the UltraSPARC T1 and UltraSPARC T2 cores under a GPL license in 2006 and 2007 [74]. The T1 and T2 processors are in-order multi-threaded eight-core processors, supporting four and eight threads respectively.

The goal of the UltraSPARC design was not to have beefy, complex, and expensive cores, but rather to have a larger number of simpler cores that can support multiple threads each to provide higher aggregate throughput. The T1 processor was produced in 2005 at 1.2 GHz in 90 nm. The whole chip was 279 M transistors in $378mm^2$ and used roughly 70 watts [66].

OpenPiton is an academic research processor built using the UltraSPARC T1 source code. Although they utilized the T1 cores (with some modifications), they implemented their own uncore and network interconnects to build a many-core processor that could scale to hundreds, if not thousands, of cores. They taped out a 25 core version on IBM 32 nm SOI process on a $36mm^2$ die with a target frequency of 1 GHz [11].

Although our discussion has currently centered on academic research, industry has their own need for RTL-level emulation of hardware designs. Engineers use "hardware emulation" platforms to test and verify new chip implementations before sending the design out for manufacturing. Even for industry, full-scale product prototypes are typically too expensive to rely on (and are often too opaque to debug as problems arise).

Unfortunately FPGAs are typically capacity-constrained and are unable to fully contain a processor design that is on the order of 100s of millions to billions of transistors. Intel demonstrated emulating a complex Intel Nehalem out-of-order core[3], but it required five Virtex-5 FPGAs, required changes to the 5% of the RTL code, and could only run at a target frequency of 520 KHz. Splitting the design to fit across an FPGA is a manual process that includes needing to implement the infrastructure to communicate across the boundaries.

[3] An 8-core Xeon Nehalem-EX is 2.3 billion transistors.

The other challenge is in translating the design to be synthesizable for FPGAs. Some constructs, such as double-phase latch RAMs and aggressive clock gating, have no direct FPGA analogue. Finally, after performing the remapping to a different implementation technology, engineers then have to reverify that the design is still the same [86].

One solution used by industry is the Palladium hardware emulation platform from Cadence Design Systems. Palladium allows engineers to directly simulate their designs without requiring RTL code changes. And to handle capacity issues, Palladium tools can seamlessly partition a single design across an entire Palladium cluster. In a blogpost from 2011, NVIDIA's Emulation Lab discussed their use of multi-million dollar Palladium XP systems to debug and verify their graphic processor (GPU) designs. Although they used multiple Palladium clusters to simulate their GPUs, the entire lab could only simulate 4 billion transistors in total [60]. For comparison, a single multimillion dollar 16-chassis Palladium emulator was used by NVIDIA to debug their Fermi microarchitecture, a 3 billion transistor GPU in 40 nm [95]. The more recent Palladium XP II platform provides support of up to 2.3 billion ASIC gates of design capacity and can emulate designs at speeds up to 4 MHz. Palladium also provides some power analysis modeling, including the ability to identify power peaks and to model low-power modes [19, 87, 20].

For established companies shipping large and complex designs, Palladium provides a productive solution for verifying designs before manufacturing. However, Palladium is not a good match for smaller teams that cannot afford the multi-million dollar price tag, or for engineers who need closer-to-real-time performance for running longer software workloads, or for early-stage design exploration where some aspects of the project may be abstracted for flexible modifications of the system under test.

2.3.3 Energy Modeling Methodologies

After the end of Denard Scaling, power consumption and energy efficiency have become a first-order design constraint. This "power wall" is further exacerbated for mobile processors, which demand desktop-level user experiences in a roughly 1 watt envelope.

Power usage is important for all environments, though for different reasons. For mobile processors, poor energy-efficiency directly limits the length of time between required battery charges. High power-usage can create thermal issues and even physically burn the user. For desktop processors, more power-usage requires louder air-cooling solutions. For servers, there is a fundamental limit to how much power can be brought physically into the building for powering the chips and the cooling systems to manage the heat given off by the processors.

Unfortunately, analyzing the power and energy usage of designs has proven to be very difficult, and the most accurate measurements can only be made after a design has been taken to the floorplanning phase (or better yet, measured from a physical chip). Instead, a common technique is to couple analytical power models with micro-architectural simulators.

Wattch is a framework at the architectural level that connects to the SimpleScalar simulator, allowed early culling of the design space via high-level power estimation. The functional simulator generates microarchitectural event counts, such as instruction fetches or cache

accesses, which are then fed into an abstract power model. However, Wattch only provides dynamic power modeling; it does not provide area, timing information, or static power modeling. As transistors have become smaller, static power usage due to increased leakage current has now become a significant fraction of total system power [15].

A similar energy estimation framework is SimplePower. SimplePower plugs into SimpleScalar and models a 5-stage pipeline, L1 caches, and off-chip memory. It uses simple analytical models for estimating memory power usage. For the busses between different levels in the memory hierarchy, SimplePower uses a transition-sensitive approach, taking into account both the switching activity and the interconnect capacitance on the bus lines. It also models transition-sensitive functional units to get the data path power usage. However, it only provides energy estimation for an integer RISC ISA using a 5-stage pipeline [118].

McPAT is the current popular tool that provides timing, area, and power estimations for multi-core and many-core processors. McPAT supports both in-order and out-of-order cores. McPAT can be integrated into any performance simulator, such as ESESC or gem5, by taking in the dynamic activity event counts and feeding them into its power models. McPAT also has the ability to model different processor energy states, and even feed that information back into the architectural simulation to model power and thermal events during the lifetime of the workload [58].

However, energy models like McPAT and Wattch can be difficult to trust. The accuracy of the tools are constrained by the fidelity of the models and are limited in the validation performed against existing systems. Component models can be too high-level, incomplete, or make incorrect assumptions about the underlying microarchitecture [123]. Regarding validation, McPAT was verified against a 90 nm Niagara, a 65 nm Niagara 2, a 65 nm Xeon Tulsa, and a 180 nm Alpha 21364 by comparing peak power numbers based on maximum switching activities. McPAT's verification saw differences of between 11% and 23% between published peak powers and McPAT's estimations [58]. Although these methodologies can be useful for exploring designs similar in technology and microarchitecture to previously verified designs, these issues are exacerbated as researchers pursue more exotic processor architectures in newer technology nodes for which there exists no good validation target.

Another issue with energy models, like the microarchitectural simulators that often feed them, is their limited simulation speed. This problem is worsened as the model becomes more detailed. However, there have been a number of techniques to accelerate energy models. One approach has been to implement the power model in a synthesizable RTL description that can be executed more quickly using an FPGA [25, 40]. Other techniques rely on an RTL-level processor description which can be executed on an FPGA to achieve high simulation speeds (10-100 MHz). However, choosing which microarchitectural events to track within the FPGA simulation is a challenge and typically requires designer intuition [13, 101].

Strober is a tool which automatically instruments an RTL-level processor design and then synthesizes and executes the design on an FPGA. Strober periodically snapshots the entire microarchitectural state from the simulation, and then loads the microarchitectural state into a *gate-level* simulation to get incredibly detailed power estimations. Designer intuition is not needed as the entire state is snapshotted. These gate-level simulations are quite accurate

as they simulate a floorplanned design at the gate and wire-level. However, they are also incredibly slow – only a few hertz. Thus Strober builds an average power estimation by sampling points in long-running workloads. Strober can also provide power estimations for any arbitrary RTL, not just designs that look like processors [56].

2.4 Processor Microarchitecture

A processor takes in a stream of instructions and performs computations as specified by each instruction. The *architecture* is the description of a processing system from the point-of-view of the software/programmer. Nearly universally, general-purpose processor architectures have consolidated on the following agreement with the programmer:

- a software program is broken down and described as a sequence of instructions
- each instruction is executed in the order specified by the program ("in-order")
- each instruction is executed one at a time before moving to the next instruction

The *microarchitecture* is an implementation of the architecture. There can be many different microarchitectures that all faithfully implement the same architecture. This distinction between the architecture and the microarchitecture was pioneered by IBM with the IBM System 360 [35]. Before the IBM S/360, new processor implementations often required significant efforts to port old software to the new machines, as program binaries were typically incompatible from one machine to the next.[4] Over time, architectures began to organically grow around different product lines (e.g., business, scientific computing, and defense) as costumers began to demand software compatibility as a requirement for buying new machines. This fracture of architectures across product categories meant that processor companies like IBM were dividing their development efforts. IBM astutely realized that the software was both more expensive and more valuable than the hardware, and that there was a benefit to unifying its disparate product lines under a single architecture.

IBM originally shipped four different microarchitectures – each at a different performance and price points – that all implemented the IBM S/360 architecture. Software written and executed on one machine could be trivially executed on a different microarchitecture. Users that wanted better performance for their software need only buy a more powerful machine. 50 years later, IBM is still shipping ever faster IBM S/360-compatible processors. The z13 microarchitecture is a 5 GHz six-core processor implemented using 4.0 billion transistors in 22 nm [120]. The S/360 Model 91 – one of the original microarchitectures released in 1966 – was only 120 thousand gates implemented using emitter-coupled logic circuits [35]. The Model 91's central CPU was composed of four "frames" – each a 6x6x1 foot cabinet composed of 20 motherboards each.

[4] One solution to binary incompatibility was the use of software simulators. However, this came with a performance overhead. Some early machines came with hardware support to speed up emulation of predecessor architectures [112].

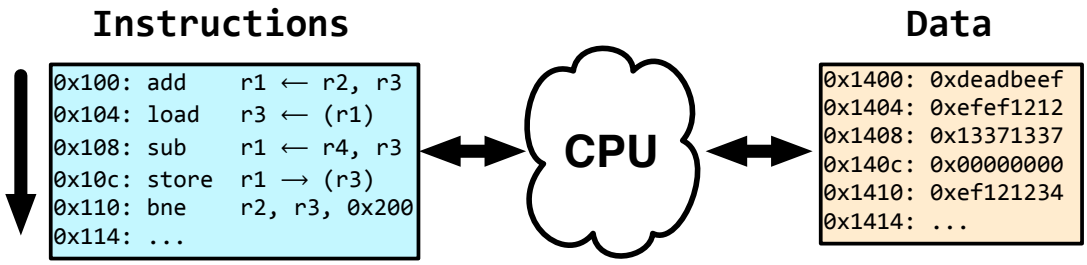


Figure 2.4: A processor takes a stream of instructions and performs computations as specified by each instruction. The hardware provides the illusion to the programmer that instructions are executed one at a time in the order specified by the program.

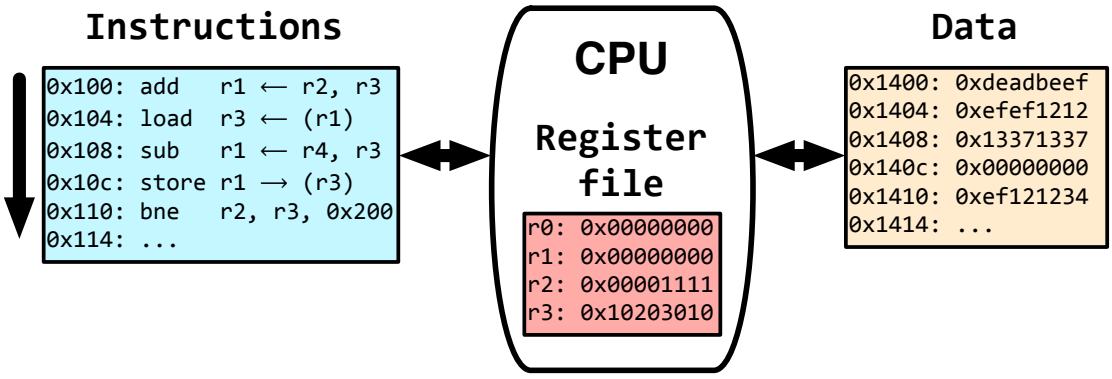


Figure 2.5: A common general-purpose processor architecture is the *register-register load/store* architecture. Working-set data is stored in a small number of *registers*. Instructions can perform logical, arithmetic, and control operations that operate on the values in the register file. Separate load and store instructions move data in to and out of the data memory.

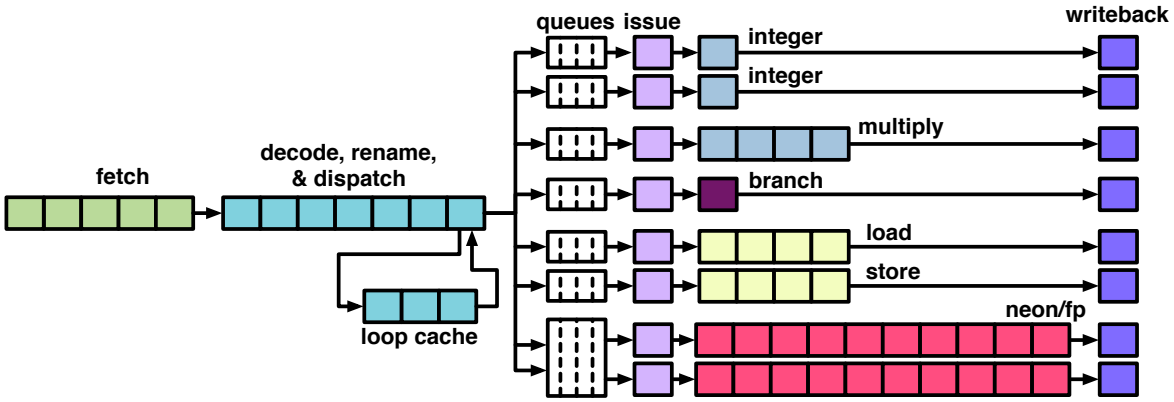


Figure 2.6: An ARM Cortex-A15 pipeline (figure adapted from [41]). The A15 illustrates the complexity of modern superscalar, out-of-order processors. Although the A15 targets mobile applications, its design is indicative of modern general-purpose application processors. See Section 3.2.1 for more details on the different stages of an out-of-order pipeline.

2.5 Out-of-order Processor Microarchitectures

A significant bottleneck in processor performance arises from long latency instructions blocking the queue of instructions behind it. For example, a *divide* instruction may take many dozen cycles to execute and produce a result. In a fully *in-order* processor pipeline, no newer instructions may be issued to the execute stage until the *divide* instruction finishes. A more common scenario is a load instruction that misses in the data cache. It may take many dozen or even hundreds of cycles before the load data returns from main memory and the load instruction can be retired.

A solution to this "head-of-queue" blocking is to allow newer instructions to proceed if they are independent of the blocked instructions. For example, an *add* instruction whose operands do not depend on the result of the long latency *divide* instruction may proceed. The *add* instruction, taking only a few cycles to execute, will finish first and write its results out-of-order with respect to the older *divide* instruction.

An even higher performance solution is to allow *any* instruction to be issued so long as its operands are available (i.e., the instruction does not depend on a still-busy instruction). Instructions that must wait on previous instructions "go to sleep" in an *Issue Window*. Newer instructions that are not dependent on any still-busy instructions, may execute immediately — or "out-of-order" with respect to the older, sleeping instructions. Sleeping instructions are woken up as their operands become available. In this manner, instructions are executed out of the Issue Window in the order that their dependences are resolved — a form of limited dataflow — and not in the order specified by the program.

Allowing instructions to be issued to the execution units out-of-order provides a more significant gain in performance. The main insight is the realization that, although a program stream is inherently serial, there is actually a significant amount of *instruction-level parallelism*. Thus, many independent instructions can be executed out-of-order with respect to one another while still maintaining the illusion of respecting the in-order instruction stream.

2.5.1 The Data-in-ROB Design (Implicit Renaming)

There are a number of ways to implement a processor with out-of-order issue. One microarchitecture is called "implicit register renaming", or "data-in-ROB". After each instruction has been fetched and decoded, it is placed in a "reorder buffer" (ROB), where all instructions inflight are tracked in-order. Each instruction then marks its destination register as "busy" in the *scoreboard*, with its ROB entry tag to denote which instruction is responsible for the busy register (the "ROB tag").

During the decode stage, each instruction reads the *scoreboard* to see if its operands are busy. If all of the instruction's operands are ready, the instruction reads its operands out of the Architectural Register File (ARF) and the instruction and its operands are placed in the ROB. The instruction is ready to be issued to the execution units at any time, even before older instructions have been issued.

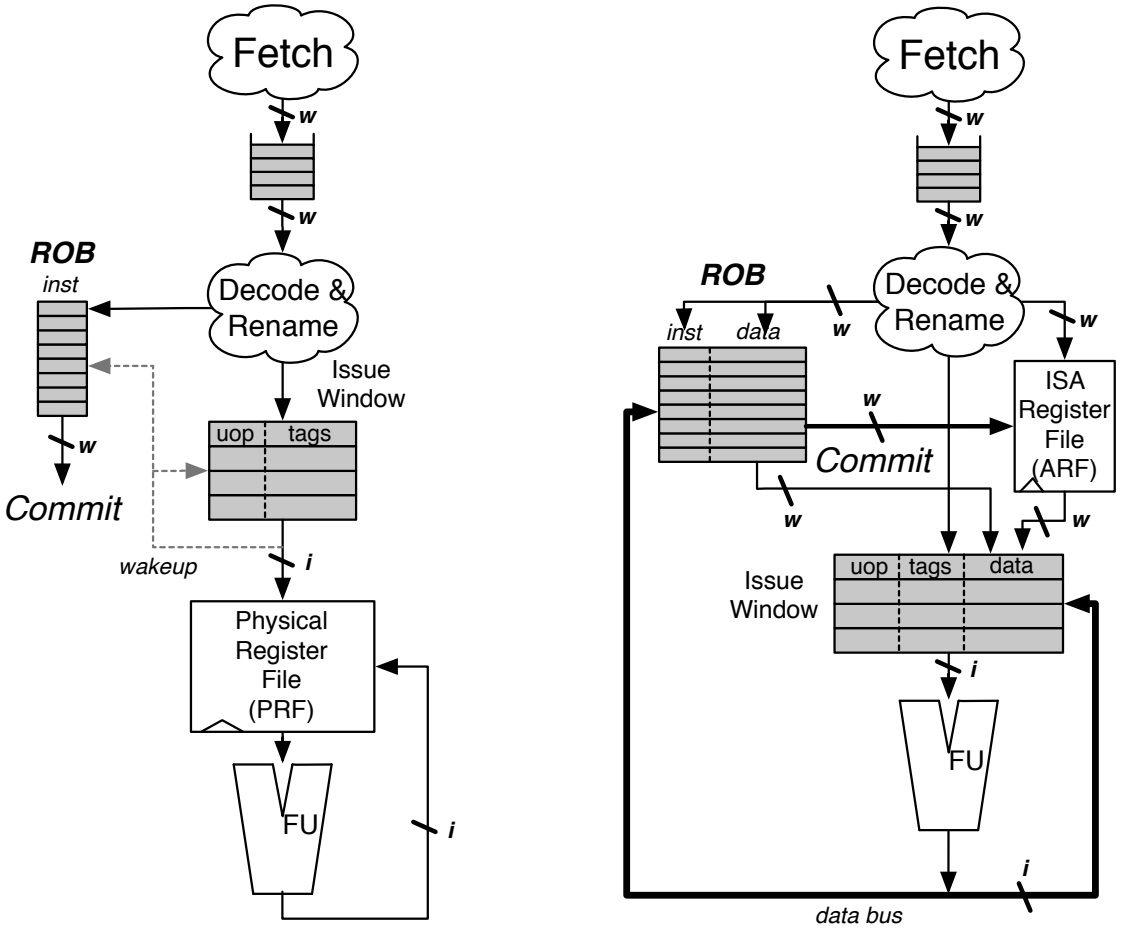


Figure 2.7: A PRF design (left) and a data-in-ROB design (right).

If an operand is marked as *busy* in the scoreboard, the instruction is unable to read its operands from the ARF. Instead, it reads from the scoreboard the ROB tags of the instructions who will generate the data values it depends on. The instruction and its operands' tags are placed in the Issue Window. The instruction then listens to the data bus; when other instructions finish executing, the functional unit broadcasts the resulting values and the corresponding ROB tags across the data bus to ROB and any dependent instructions waiting in the Issue Window. Once the instruction has all of its operands, it may request to be issued to the execution units.

In this design, the Architectural Register File (ARF) only holds the committed register state while the ROB holds the speculative write-back data. On commit, the ROB transfers the speculative data to the ARF. The Pentium 4 is an example of an *implicit renaming* design.

2.5.2 The Physical Register File Design (Explicit Renaming)

An other style of out-of-order execution, which we will label the Physical Register File (PRF) design, uses "explicit register renaming".

A *physical register file*, containing many more registers than the ISA dictates, holds both the committed architectural register state and the speculative register state. This is in contrast to the data-in-ROB design, in which committed architectural state resides in the ARF and speculative data resides in the ROB.

During the rename stage, the renamer maps the *ISA* (or *logical*) register specifiers of each instruction to *physical* register specifiers. A *map table* tracks the mappings from *logical* registers to *physical* registers. When an instruction needs to write a register, it is allocated a new physical register. If the instruction is misspeculated, the previous mapping is restored. As instructions execute, they write back their values immediately to the PRF. Newer instructions can then read their operands out of the PRF. If an exception occurs, the rename map table contains the information needed to recover the committed state.

An advantage of this style is that all data is written into and read from a single PRF – there is no broadcasting of all write data on a common data bus across to all issue windows.

The MIPS R10k [124], Alpha 21264 [55], Intel Sandy Bridge, and ARM Cortex-A15 cores are all example of explicit renaming out-of-order cores.

2.5.3 The Differences Between the Two Styles

Figure 2.7 and Table 2.5 show a side-by-side comparison of the two styles of out-of-order processors. Each style makes different tradeoffs.

The data-in-ROB design uses a architectural register file (ARF), only storing the committed data values for each ISA register. This is fewer registers than the PRF design, whose register file must store both the committed architectural state and the speculative architectural state. However, the ARF must provide enough register file read ports to supply operands to instructions as they are decoded. For a 4-wide decode stage, a machine that

Table 2.5: The differences between the data-in-ROB design and the physical register file design.

	Data-in-ROB Design	Physical Register File Design
committed data	Architectural Register File	Physical Register File
speculative data	ROB	Physical Register File
operand read	Decode Stage	Register File Read Stage (after Issue Stage)
write-back	writing data into ROB	writing data into PRF
wakeup	broadcasting data to Issue Window	broadcasting register pointers to Issue Window
committing	writing data into ARF	updating rename table

supports three-operand instructions must provide up to 12 read ports. An worst-case scenario would be heavy-computational code that features a significant density of fused multiply-add instructions (FMAs). One solution is to limit the maximum number of reads per cycle, but this directly limits instruction bandwidth and would hurt performance of densely packed arithmetic codes. Meanwhile, the write port count is tied to the machine's commit width.

The PRF design does not suffer the same read port limitation. Instead, it can match the read port requirements to the number of functional units supported. It can even rely on dynamic read port scheduling and speculate that most instructions do not require more than one operand [110]. A hypothetical 4-issue processor with two integer units, one floating point FMA unit, and one load/store unit (and assuming a unified integer/floating point register file) would need at most 9 read ports to fully satisfy all of its functional units.[5] However, by recognizing that many instructions only use one operand, and using dynamic read port scheduling, this hypothetical processor could use even fewer read ports. The number of write ports used in a PRF design is proportional to the number of functional units.

2.6 History of Out-of-order Processors

The CDC 6600 (1964) was the first machine to use an out-of-order issue pipeline. In the central processor, the core maintained a *scoreboard* of busy registers and functional units. If an instruction did not depend on a *busy* register and the functional unit was available, the instruction could be issued out-of-order with respect to older, stalled instructions. The processor stalled instructions:

1. if its operand was busy (a read-after-write (RAW) hazard, or *true dependence*).
2. if no functional unit was available.
3. if the instruction would overwrite a busy register (a write-after-write (WAW) hazard, or *output dependence*).

As the CDC 6600 had ten different functional units and 24 registers, it was able to exploit a modest amount of instruction-level parallelism by executing up to ten independent instructions simultaneously [108].

In 1966, the IBM System/360 Model 91 introduced the data-in-ROB design for its floating point unit. The data-in-ROB design allowed for a limited amount of "implicit register renaming"; the architecture only provided four floating-point registers, but speculative results could be stored in the ROB and forwarded to newer instructions. Register renaming allowed the Model 91 to execute past write-after-read (WAR) and WAW hazards. These hazards are a common obstacle to executing multiple loop iterations simultaneously as the same register specifiers are encountered each loop iteration.

[5]For a processor supporting an ISA like RISC-V, only the floating point unit has 3-read instructions.

However, out-of-order execution came with a number of challenges that increased complexity and limited potential performance. One of the early challenges was speculating past branches.[6] Without the ability to accurately speculate branches, the processors were unable to fetch enough instructions to keep the pipeline fully occupied. Supporting restartable exceptions that maintained the programmer's illusion of in-order program execution was another complexity. Therefore, out-of-order designs receded until 1990, when IBM released the first out-of-order micro-processor, the POWER1. The floating point unit utilizes register renaming to help overlap floating point arithmetic operations from the current loop iteration with floating point memory operations from the next loop iteration [42].

Since the POWER1, out-of-order processors have become the popular microarchitectural choice for general-purpose computing needs. Single-threaded performance of general-purpose applications is dominated by out-of-order processors [98]. In the datacenter (which service many parallel requests), out-of-order processors are chosen over more energy-efficient in-order cores due to their ability to better meet service-level agreement guarantees [45]. Cell phones have also recently adopted out-of-order processors, such as the iPhone 4s, released in 2011, which uses an 800 MHz ARM Cortex-A9.

2.7 The Value of Out-of-order Execution

Although there are a number of different styles of architectures and microarchitectures, out-of-order processors are the popular choice for providing the best performance for general-purpose applications.

The first advantage of out-of-order (OOO) microarchitectures, coupled with aggressive speculation mechanisms, is their ability to construct superior instruction schedules. OOO performance is largely constrained only by the true dependencies between instructions and functional unit availability. By leveraging sophisticated speculation techniques, OOO cores can speculate past control hazards and memory ordering hazards to develop an aggressive schedule of instructions that is unavailable to in-order cores.

As one example, OOO cores rely on register renaming to break the anti- and output-dependencies between loop iterations. This allows OOO cores to dynamically unroll loops and execute multiple loop iterations in parallel simultaneously. In-order (IO) cores are largely unable to overlap successive loop iterations, as they must wait for the registers used in one iteration to free up before they can be reused in the next iteration. IO cores could adapt register renaming to provide each loop iteration with a new pool of physical registers to use, but the added complexity and hardware costs of register renaming make the marginal increase in complexity in adding out-of-order issue relatively small. Another possible strategy for IO cores is to rely on the compiler to statically unroll the loop. However, this comes with its own costs, including increased code size, and can exhibit poor performance-portability across different IO designs.

[6] The S/360 model 91 speculated the branch was taken if the branch target was one of the last 8 instructions executed.

Another example of superior code scheduling is the OOO core's ability to handle unbiased branches. OOO cores can speculate past branches and effectively "hoist" instructions out of a basic block and begin execution before the branch is resolved. Although an optimizing compiler can effectively mimic the same behavior, it must statically choose which path through the program to optimize for. However, branches are typically far more predictable than they are biased. For example, a completely unbiased branch may alternate between taken and not-taken paths in an easily predicted manner; the OOO design can perfectly schedule this code sequence while an optimizing compiler attempting to statically schedule code for an IO core will be utterly helpless.

The second advantage to OOO microarchitectures is that the same instructions in a program can be executed using a different schedule from one iteration to the next, as a response to dynamic events such as cache misses or branch mispredictions. This advantage is known as *dynamism*, and it provides OOO designs with the flexibility to maintain a full pipeline despite suboptimal static code schedules or unpredictable, long latency events such as cache misses or branch mispredictions. Meanwhile, IO designs are more susceptible to performance cliffs created by suboptimal instruction scheduling.

One example of dynamism at play is the better branch recovery exhibited by OOO designs. OOO designs can overlap the branch recovery with the critical path of instructions from before the branch. An IO design is at the mercy of the critical path latency assumptions built into the code schedule, but an OOO design executes strictly based on instruction readiness [64].

By dynamically finding and exploiting instruction-level parallelism within a program — executing instructions in the order that the data is available, rather than sticking to an overly strict, sequential ordering of instructions — OOO designs relieve compilers and programmers of the pressure to provide good code schedules. However, some of these advantages are not entirely restricted to OOO cores. A number of attempts have been made to provide at least some of the advantages of OOO microarchitectures to simpler, more energy-efficient designs. However, despite efforts to improve IO performance — adding run-ahead execution, data prefetching, register checkpointing, among others — IO cores have still been unable to fully match the performance of OOO microarchitectures [64].

2.8 Conclusion

Due to the trends in technology, single-threaded performance is becoming more difficult to improve, and power has become a first-order design constraint. But despite the technology trends and slowing of single-threaded performance, workloads continue to evolve. Unfortunately, software simulators struggle to evaluate long-running and complex workloads and are often unable to provide the fidelity to confidently evaluate ideas that provide important, but small gains. Single-thread performance is still important and a bottleneck even for large, multi-threaded systems. Despite their complexity, out-of-order processors are the best microarchitecture for dynamically exploiting instruction-level parallelism and providing

excellent single-thread performance. For these reasons, this thesis will explore implementing an industry-competitive, high-performance out-of-order processor.

Chapter 3

BOOM Overview

The goal of this chapter is to provide an overview of the design and implementation of the *Berkeley Out–of–Order Machine* (BOOM). Out-of-order processors are the microarchitecture du jour, used to provide high-performance general-purpose computing across a wide spectrum of platforms, from mobile processors up to the main compute nodes in warehouse-scale computers. The goal of BOOM is to provide a synthesizable, prototypical baseline processor for research, education, and industry. As part of that mission, we fabricated an SRAM resiliency chip using BOOM as its processing core. Although simpler cores have been previously used for these SRAM resiliency studies [54, 127], BOOM may provide a more accurate analog to modern processors. In particular, the out-of-order scheduling of BOOM will allow for a better tolerance of the longer average latencies of memory operations brought about by the dynamically reduced cache capacity when running in a lower-power mode.

Section 3.2 provides an overview of the microarchitecture of BOOM. Section 3.3 discusses the design methodology and how using a new hardware language called *Chisel* provided us a big productivity win. Section 3.4 touches on the FPGA simulation methodology that we were able to leverage due to the synthesizability of BOOM. Section 3.5 goes into greater detail on the VLSI implementation effort to tape out a processor using BOOM with a two-graduate student team. Although BOOM is a parameterizable generator, Section 3.6 discusses the specific configuration we chose for a tape-out of BOOM, which is representative of the parameters we focused on optimizing. We then conclude our overview of BOOM with Section 3.7, which briefly provides performance comparisons of BOOM in relation to a selection of contemporary industry processors.

3.1 The RISC-V Instruction Set Architecture

BOOM implements the RV64G variant of the RISC-V Instruction Set Architecture (ISA). RV64G denotes the 64 bit address variant of the "general-purpose" subset. This subset includes the standard MAFD extensions which cover integer multiply/divide, atomic memory operations, load-reserve/store-conditional, single- and double-precision IEEE 754-2008 float-

ing point. BOOM also implements the Privileged ISA specification v1.9 and the External Debug Support specification v0.11.

RISC-V is a free and open ISA developed at UC Berkeley to provide a clean, simple, and high-performance architecture for use in research, education, and industry. Although inspired by previous MIPS-based research derivatives, RISC-V is a "clean break" implementation. RISC-V is now managed by the RISC-V Foundation, a non-profit trade group composed of more than sixty member companies. For many companies, RISC-V provides an opportunity to avoid vendor lock-in to any particular processor company. It also provides companies the freedom to build their own RISC-V processors without requiring licensing agreements from the controlling company of the ISA [9].

For researchers, RISC-V's simplicity and lack of historical baggage allows them to focus on their innovative ideas without distraction from a complex and bloated architecture. RISC-V's G subset provides the following features that make it easy to target with high-performance designs:

Accrued floating-point exception flags The FP status register does not need to be renamed, nor can FP instructions throw exceptions themselves.

No integer side-effects All integer ALU operations exhibit no side-effects, save the writing of the destination register. This prevents the need to rename additional condition state.

No cmov or predication Although predication can lower the branch predictor complexity of small designs, it greatly complicates OOO pipelines, including the addition of a third read port for integer operations.

no implicit register specifiers Even JAL (jump-and-link) requires specifying an explicit destination register `rd`. This simplifies rename logic, which prevents either the need to know the instruction first before accessing the rename tables, or it prevents adding more ports to remove the instruction decode off the critical path.

Register specifiers `rs1`, `rs2`, `rs3`, `rd` are always in the same place This regularity allows decode and rename to proceed largely in parallel.

Relaxed memory model This decision greatly simplifies the Load/Store Unit, which does not need to have loads snoop other loads nor does coherence traffic need to snoop the LSU, as required by stricter memory models such as *total store order* or *sequential consistency*.[1]

[1] A RISC-V Foundation Working Group is analyzing proposals to strengthen the relaxed memory model that will likely require ordering loads to the same address.

3.2 The BOOM Microarchitecture

3.2.1 The BOOM Pipeline

Conceptually, BOOM is broken up into 10 stages: *Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback,* and *Commit*. The *Commit* stage occurs asynchronously with regards to the rest of the pipeline. Figure 3.1 shows a simplified high-level outline of the BOOM pipeline.

Fetch Instructions are *fetched* from the Instruction Memory and pushed into a FIFO queue, known as the *fetch buffer*.

Decode *Decode* pulls instructions out of the *fetch buffer* and generates the appropriate "micro-op" to place into the pipeline.

Rename The ISA, or "logical", register specifiers are then *renamed* into "physical" register specifiers.

Dispatch The micro-op is then *dispatched*, or written, into the *Issue Window*.

Issue Micro-ops sitting in the Issue Window wait until all of their operands are ready and are then *issued*. This stage is the beginning of the out–of–order part of the pipeline.

RF Read Issued micro-ops first *read* their operands from the unified physical register file (or from the bypass network)...

Execute ... and then enter the *Execute* stage where the functional units reside. Issued memory operations perform their address calculations in the *Execute* stage, and then store the calculated addresses in the Load/Store Unit that resides in the *Memory* stage.

Memory The *Load/Store Unit* consists of three queues: a *Load Address Queue* (LAQ), a *Store Address Queue* (SAQ), and a *Store Data Queue* (SDQ). Loads are fired to memory when their address is present in the LAQ. Stores are fired to memory at *Commit* time (stores cannot be *committed* until both their address and data have been placed in the SAQ and SDQ).

Writeback ALU operations and load operation results are *written* back to the physical register file.

Commit The *Reorder Buffer*, or ROB, tracks the status of each instruction in the pipeline. When the head of the ROB is not-busy, the ROB *commits* the instruction. For stores, the ROB signals to the store at the head of the Store Queue that it can now write its data to memory.

Some of these stages are combined together when possible, while other stages are further sub-divided to achieve reasonable clock frequencies.

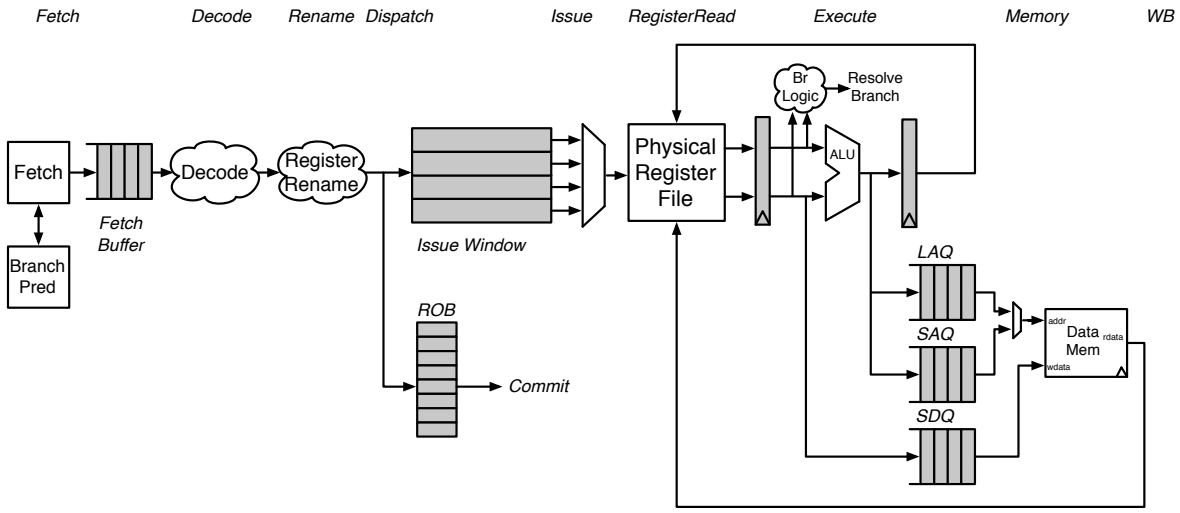


Figure 3.1: A conceptual outline of the BOOM pipeline.

3.2.2 Instruction Fetch and Branch Prediction

The purpose of the *frontend* is to fetch instructions for execution in the *backend*. Processor performance is best when the frontend provides an uninterrupted stream of instructions. In order to maintain an uninterrupted stream of instructions, the frontend must utilize branch prediction techniques to predict which branch path it believes the instruction stream will take long before the branch's direction can be properly resolved. Any mispredictions in the frontend will not be discovered until the branch (or jump-register) instruction is executed later in the backend. In the event of a misprediction, all instructions after the branch must be flushed from the processor and the frontend must be restarted using the correct instruction path.

The frontend relies on a number of different branch prediction techniques to predict the instruction stream, each trading off accuracy, area, critical path, and pipeline penalty when making a prediction.

3.2.2.1 Branch Target Buffer (BTB)

The BTB maintains a set of tables mapping from *instruction addresses* (PCs) to *branch targets*. When a lookup is performed, the *look-up address* indexes into the BTB and looks for any *tag matches*. If there is a *tag hit*, the BTB will make a prediction and may redirect the frontend based on its predicted *target address*. Some *hysteresis* bits are used to help guide the *taken/not-taken* decision of the BTB in the case of a *tag hit*.

3.2.2.2 Return Address Stack (RAS)

The RAS predicts function returns. *Jump-register* instructions are otherwise quite difficult to predict, as their target depends on a register value. However, functions are typically entered using a *Function Call* instruction at address A and return from the function using a *Return* instruction to address $A+1$.[2] The RAS can detect the call, compute and then store the expected return address, and then later provide that predicted target when the *Return* is encountered. To support multiple nested function calls, the underlying RAS storage structure is a stack.

3.2.2.3 Conditional Branch Predictor (BPD)

BOOM implements a *global history* predictor to predict the direction of conditional branches. Global history predictors work by tracking the outcome of the last N branches in the program (providing a "global" view) and hashing this *history* with the *look-up address* to compute a look-up index into the BPD prediction tables.

The BPD *only* makes *taken/not-taken* predictions; it therefore relies on some other agent to provide information on what instructions are branches and what their targets are. The

[2] Actually, it will be $A+4$ as the size of the call instruction to jump over is 4 bytes in RV64G.

BPD can either use the BTB for this information or it can wait and decode the instructions themselves once they have been fetched from the instruction cache. Because the BPD does not store the expensive branch targets, it can be much denser and thus make more accurate predictions on the branch directions than the BTB; whereas each BTB entry may be 60 to 128 bits, the BPD may be as few as one or two bits per branch.

3.2.3 The Decode Stage and Resource Allocation

The decode stage takes instructions from the fetch buffer, decodes them, and allocates the necessary resources as required by each instruction (for example, loads are allocated an entry in the Load Address Queue). The decode stage will stall as needed if not all resources are available.

3.2.4 The Register Rename Stage

Renaming is a technique to rename the ISA (or *logical*) register specifiers in an instruction by mapping them to a new space of *physical* registers. The goal of *register renaming* is to break the output- (write-after-write) and anti-dependences (write-after-read) between instructions, leaving only the true dependences (read-after-write). This technique is critical to allowing out-of-order processors to speculatively execute far ahead of the commit point. For example, output- and anti-dependency hazards are encountered when executing loops. With renaming, processors can "unroll" the loop in hardware and execute multiple loop iterations simultaneously, bounded only by the amount of true dependencies across loop iterations and the issue width of the processor.

BOOM is a *physical register file* out-of-order core design. A physical register file, containing many more registers than the ISA dictates, holds both the committed architectural register state and speculative register state. The rename map tables contain the information needed to recover the committed state. As instructions are renamed, their register specifiers are explicitly updated to point to physical registers located in the physical register file.

When a branch passes through the Rename stage, a copy of the map table is made. On a branch mispredict, the map table can be reset instantly from the mispredicting branch's copy of the map table.

3.2.5 The Reorder Buffer (ROB) and Exception Handling

The ROB tracks the state of all inflight instructions in the pipeline. The role of the ROB is to provide the illusion to the programmer that their program executes in-order. If a misspeculation, exception, or interrupt occurs, the ROB is tasked with helping reset the microarchitectural state.

After instructions are decoded and renamed, they are then dispatched to the ROB and the issue window and marked *busy*. As instructions finish execution, they inform the ROB and are marked *not-busy*. Once the *head* of the ROB is no longer busy, the instruction is

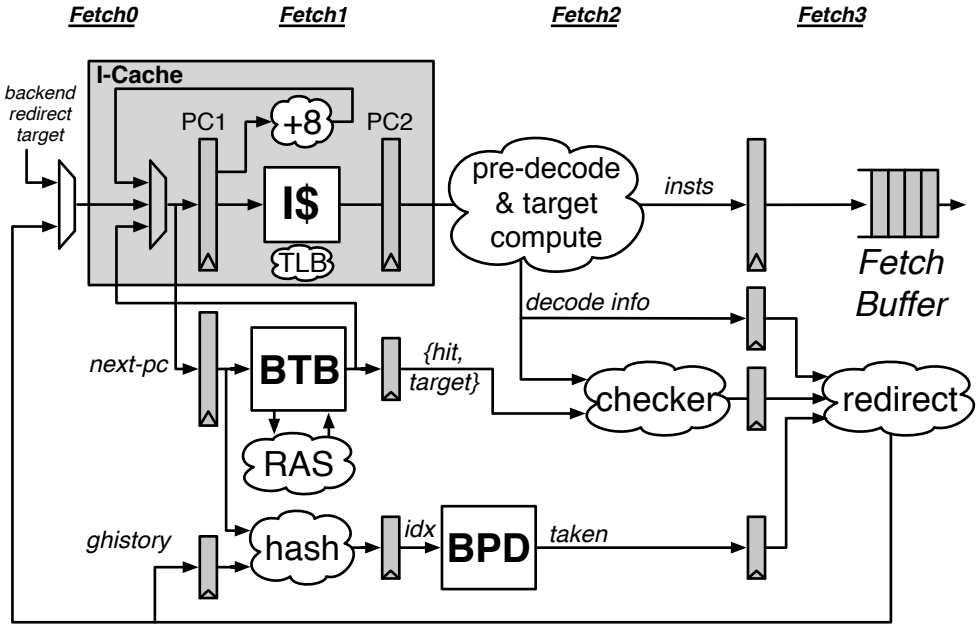


Figure 3.2: The instruction fetch frontend to BOOM. The BTB predictions are performed in a single cycle, while the more complex conditional branch predictor takes three cycles. As the BTB predictions are made using partial tags, a *checker* module is required to verify that the predicted target is valid. If no prediction is asserted, the instruction cache will continue to fetch the next 8 bytes (for a two-instruction-wide frontend).

committed, and its architectural state is now visible. If an exception occurs and the excepting instruction is at the head of the ROB, the pipeline is flushed and no architectural changes that occurred after the excepting instruction are made visible. The ROB then redirects the frontend to begin fetching instructions from the appropriate exception handler. During an exception, the rename map tables are restored by rolling back the ROB over a period of many cycles.

3.2.6 The Issue Unit

The issue window holds dispatched micro-ops that have not yet executed. When all of the operands for the micro-op are ready, the issue slot sets its "request" bit high. The issue select logic then chooses to issue a slot that is asserting its request signal. Once a micro-op is issued, it is removed from the issue window to make room for more dispatched instructions. BOOM uses three separate issue windows, partitioned on micro-op type (integer, floating point, memory).

3.2.7 The Register File and Bypass Network

BOOM is a unified, physical register file design. The register file holds both the committed and speculative state. Separate register files are used for storing integer and floating-point register values.

The register file statically provisions all of the register read ports required to satisfy all issued instructions.

ALU operations can be issued back-to-back by having the write-back values forwarded through the bypass network. Bypassing occurs at the end of the *Register Read* stage.

3.2.8 The Execution Pipeline

The Execution Pipeline contains many different functional units to handle different operation types: arithmetic operations, divides, address calculations, and floating-point manipulations. The Execution Pipeline contains many different types of functional units, most pipelined, like the fused multiply-add unit, while others, like the divider, are iterative. The *issue select* ports on the issue windows must intelligently schedule micro-ops onto the functional units while honoring their occupancy and hazard requirements. Chapter 5 describes in greater detail how the Execution Pipeline is described in a generalizable manner to allow for a good deal of parameterization and reduced developer burden.

3.2.9 The Load/Store Unit (LSU)

The Load/Store Unit is responsible for deciding when to fire memory operations to the memory system. There are three queues: the Load Address Queue (LAQ), the Store Address Queue (SAQ), and the Store Data Queue (SDQ).

Once a store instruction is committed, the corresponding entry in the Store Queue is marked as committed. The store is then free to be fired to the memory system at its convenience. Stores are fired to the memory in program order.

Loads are optimistically fired to memory on arrival to the LSU. Simultaneously, the load instruction compares its address with all of the store addresses that it depends on. If there is a match, the memory request in the cache is killed. If the corresponding store data is present, then the store data is *forwarded* to the load and the load marks itself as having *succeeded*. If the store data is not present, then the load goes to *sleep*. Loads that have been put to sleep are retried at a later time. If a load incorrectly executes ahead of an older store to the same address, it will read out stale data. In this scenario, the entire pipeline must be flushed and the load refetched once it reaches the commit point.

BOOM implements the *release consistency* memory model [39] described by the RISC-V User-Level ISA v2.2, in which stores and loads may be freely re-ordered but synchronization primitives with *release* and *acquire* semantics receive stronger ordering.

More explicitly, BOOM exhibits the following behavior:

- Write→Read constraint is relaxed (newer loads may execute before older stores).
- Read→Read constraint is relaxed (loads to the same address may be reordered).
- A thread can read its own writes early.

Naturally, a newer load may not execute before a store to the same address.

3.2.10 The Memory System

BOOM uses a non-blocking data cache. The cache has a three-stage pipeline and can accept a new request every cycle.

- S0: Send request address
- S1: Access SRAM
- S2: Perform way-select and format response data

The data cache is also cache coherent which is helpful even in uniprocessor configurations for allowing a host machine or debugger to read BOOM's memory.

3.3 Design Methodology

3.3.1 RTL Design

BOOM is implemented at the register-transfer level (RTL) in the *Chisel* hardware construction language (see Section 3.3.3). *Chisel* generates a Verilog representation of BOOM which

can then be fed into standard VLSI tools for synthesis, place, and route. We repeatedly used results gathered from synthesis, and less often data from place-and-route, to drive design decisions and to guide small changes, both implemented in the *Chisel* RTL.

This continuous feedback allowed us to quickly adapt to issues that showed in the VLSI reports. We did not "freeze" RTL before the tape-out, but instead had multiple parallel synthesis and place-and-route jobs running with fresh changes to the RTL. The use of *Chisel* does not preclude pre-placing and floor-planning of any blocks. We also relied on register retiming within the VLSI synthesis tools to facilitate a more flexible *Chisel* RTL description. Instead of having to painfully place registers at the optimal point in the design, the synthesis tools could be trusted to move registers to wherever provided the best *quality of result*. Synthesis was roughly two to three hours while place-and-route took roughly 14 hours for heavily congested designs and down to 9 hours once we had improved BOOM's critical path and congestion problems.

3.3.2 RTL Verification

Verification of BOOM's RTL was performed using full-system testing by executing programs on cycle-exact simulators of BOOM. Verilator [117], a fast and open-source Verilog simulator, was used to transform the *Chisel*-generated Verilog into a cycle-exact simulator. We also used the proprietary Synopsys Verilog Compiler Simulator (VCS) for RTL and gate-level simulations. Although VCS compiles noticeably faster than Verilator when full waveform debug mode is enabled (4 minutes versus 21 minutes), VCS simulates BOOM at roughly 1/20 the speed of Verilator. When using the parameters shown in Table 3.4, Verilator simulates BOOM at 9,980 Hz and VCS simulates BOOM at 490 Hz (as measured running `Hello World`). Table 3.1 shows a more complete breakdown of the compile and simulation speeds of BOOM. All development work was done on a 12-core Intel Xeon E5-2643 v2 running at 3.50 GHz.

The basic sets of functional tests executed on BOOM is the `riscv-tests` suite [81], which contains a set of 232 hand-written assembly programs that test instruction functionality with limited white-box testing and 11 bare-metal C benchmarks (see Table 3.2). These test simulations take roughly five minutes to run using Verilator across 12 Intel Xeon cores. We also run the Coremark benchmark and Hello World on the RISC-V proxy kernel (`riscv-pk`) to test running user-level programs on top of a privileged architecture. To provide more confidence of correctness, but no debug visibility, we used an FPGA instantiation of BOOM running at 50 MHz to test the ability to boot Linux and run applications that run for minutes to hours in *target* time.

We use `riscv-torture` [82] to perform adversarial torture testing. The `riscv-torture` program generates a random sequence of assembly instructions, pulled from a library of assembly snippets and then interleaved. Although `torture` was able to find a number of bugs early in BOOM's development, `torture` has some significant blind spots that made it less useful for finding more difficult-to-exercise bugs. For one, it does not exercise any privileged architecture. Second, all branches are only executed once — `torture` does not

Table 3.1: The compile and simulation times for the Verilator and VCS RTL simulators when built using a 12-core Intel Xeon E5-2643 v2 (3.5 GHz) with parallelized `make` enabled. The Chisel RTL must first be compiled to the FIRRTL intermediate representation language, and then from FIRRTL the RTL can be lowered into Verilog. Verilator and VCS then consume the Verilog RTL description to create a cycle-accurate simulator of BOOM.

Compilation Step	Compile Time		Simulation Time
Chisel → FIRRTL	0 m	12 s	n/a
FIRRTL → Verilog	1 m	4 s	n/a
Verilog → Verilator	3 m	3 s	9 980 Hz
Verilog → Verilator (debug)	20 m	13 s	6 310 Hz
Verilog → VCS	1 m	59 s	490 Hz
Verilog → VCS (debug)	1 m	59 s	370 Hz

generate loops. Many of BOOM's more challenging bugs rely on the building up and killing of significant speculative state, which is most often exercised when running multiple loop iterations simultaneously and then mispredicting the loop exit.

To find these more challenging bugs that slip past `riscv-tests` and `riscv-torture`, we rely on two additional techniques. The first is the use of *assertions* – a code statement that must always be *true*. Written in *Chisel*, these assertions stop simulation with a failure code if a bad processor state has been encountered such that the assertion is no longer *true*. An example is found in BOOM's re-order buffer (ROB): when an instruction finishes executing and informs the ROB that it is no longer busy, an assertion checks that the unbusied ROB entry contains a valid instruction and that the ROB entry was previously *busy*. If the instruction is trying to unbusy an invalid entry or an already *not-busy* entry, then an invalid state has been encountered and the simulation is stopped with a failure code. One downside of these assertions is that they are unsynthesizable constructs and thus are only available in simulation mode.

The second technique for finding bugs that has demonstrated significant success is comparing BOOM's instruction commit log against `spike`'s instruction commit log. These commit logs contain a sequence of the committed instructions' PC address, instruction encoding, register writeback address, and register writeback data. However, there are some significant challenges with this type of co-simulation debugging. The first is that there are valid differences between `spike` and BOOM that must be addressed — store-conditionals, timer interrupts, and micro-architectural counters. The second challenge is that `spike` and BOOM must present the *exact* same platform model; for example, differences in memory sizes or available drivers could cause deviations in the resulting simulation. Like assertions, the commit log capabilities are only available in simulation, which reduces the types of workloads that can be tested. However, ongoing research is exploring making both the assertions and commit log capabilities synthesizable constructs for FPGA simulations, which would allow both assertions and the commit log tracing to be used in a high-speed environment for verifying longer running programs at roughly 50 to 100 MHz [8].

3.3.3 The *Chisel* Hardware Construction Language

BOOM is implemented in the *Chisel* hardware construction language (HCL). *Chisel* was developed by UC Berkeley to enable advanced hardware design using highly parameterized generators. *Chisel* allows designers to utilize concepts such as object orientation, functional programming, parameterized types, and type inference. UC Berkeley implemented *Chisel* as an open-source HCL built on top of the Scala language. However, *Chisel* is *not* a "high-level synthesis" language; *Chisel* users still reason about registers, wires, and memories. The BOOM source-code repository is 16,000 lines of *Chisel* code.

During elaboration time, *Chisel* source code is compiled into an intermediate representation known as *FIRRTL* (Flexible Intermediate Representation for RTL). The *FIRRTL* compiler then gradually lowers the *Chisel*-generated *FIRRTL* code into Verilog code through a series of compiler transformations. This explicit split between the *Chisel*-frontend and

Table 3.2: The set of bare-metal benchmarks provided by the `riscv-tests` repository. Each is on the order of 100k instructions.

dhrystone	A synthetic embedded integer benchmark.
median	Performs a 1D three element median filter.
mm	Performs a floating-point matrix multiply.
mt-matmul	A multi-threaded implementation of `mm`.
mt-vvadd	A multi-threaded implementation of `vvadd`.
multiply	A software implementation of multiply.
qsort	Sorts an array of integers using quick sort.
rsort	Sorts an array of integers using radix sort.
spmv	Double-precision sparse matrix-vector multiplication.
towers	Solves the Towers of Hanoi puzzle recursively.
vvadd	Sums two arrays and writes into a third array.

FIRRTL-backend allows the hardware designer access to the compiler internals in such a way as to make it easier for designers to add their own transformation passes. For example, ASIC tool flows and FPGA tool flows both consume Verilog, but the treatment of some primitives such as memories may differ significantly. *FIRRTL* provides the options to instantiate behavioral models of memories — useful for FPGA backends — or it can automatically blackbox memories to facilitate designers providing their own memory blocks (common in ASIC methodologies).

3.3.4 The *Rocket-chip* System-on-a-Chip Generator

BOOM's source code only describes a core; the rest of the memory system, uncore, and remaining SoC infrastructure must be provided.

As such, BOOM was developed to use the open-source *Rocket-chip* SoC generator [7]. The *Rocket-chip* generator can instantiate a wide range of SoC designs, including cache-coherent multi-tile designs, cores with and without accelerators, and chips with or without a last-level shared cache.

Rocket-chip is built around the Rocket in-order core. Rocket is a single-issue 5-stage in-order core that implements the RV64GC ISA and page-based virtual memory. The original design purpose of the Rocket core was to enable architectural research into vector co-processors by serving as the scalar *Control Processor*. *Rocket-chip*, using Rocket as its core, has been taped out at least fourteen times in three different commercial processes, and has been successfully demonstrated to reach over 1.65 GHz in IBM 45 nm SOI [100].

From BOOM's point of view, *Rocket-chip* can also be thought of as a "library of processor components." There are a number of modules created for Rocket that are also used by BOOM: the functional units, the caches, the translation look-aside buffers, the page table walker, the debug transport module, the privileged architecture state, and more. *Rocket-chip* is implemented in 38,500 lines of *Chisel* code. By leveraging the existing silicon-proven *Rocket-chip* infrastructure, we have been able to maintain focus on the BOOM core itself.

3.4 FPGA Implementation

Field-programmable Gate Arrays (FPGAs) are specialized integrated circuits that can be programmed to emulate arbitrary hardware functions. The basic building block of an FPGA is the *configurable logic block* (CLB). Each logic block is built out of *look-up tables* (LUTs), memory state elements, and simple logic operations such as full-adders and multiplexer selectors. Hundreds of thousands of logic blocks are then tiled across the FPGA. Both the CLBs and the wiring connecting CLBs together are configurable by writing to configuration memories to govern their behavior. In this manner, arbitrary hardware functions can be executed on an FPGA. FPGAs also provide more fixed-function blocks, such as floating-point arithmetic units, to more efficiently target particular applications. The hardware functions being emulated by modern FPGAs can be reconfigured repeatedly.

As FPGAs have grown rapidly in size, they have increasingly become attractive options for emulating processor designs before committing to a particular design to send to the foundry for fabrication.

We use the Xilinx ZC706 FPGA to emulate instances of BOOM. Synthesis, place, and route take approximately 45 minutes to 60 minutes on an Intel Xeon Ivybridge. BOOM runs at 40 to 50 MHz, a considerable speedup of roughly 5,000x - 100,000x over software simulation of the same BOOM design using Verilator (10 kHz) or VCS (500 Hz). By using Xilinx's Memory Interface Generator (MIG), we are able to synthesize a DDR3 memory controller to communicate with 8 GB of DDR3 SODIMM Memory that is on the ZC706 board. We run BOOM in a tethered simulation with a hardened ARM Cortex-A9 running a front-end server to manage the host/target communications, which include binary loading and console I/O.

By using an FPGA to simulate BOOM, running the full reference input sizes to benchmarks becomes feasible. For example, one trillion cycles takes roughly 6 hours of wall clock time. As SPECint2006 is roughly 20 trillion instructions spread across 35 workloads, a design able to achieve one instruction per cycle can finish SPEC with reference inputs in five days. With a small cluster of FPGA boards, this can be sped up to about 18 hours, the length of time of the longest running workload.

3.5 ASIC Implementation

On Aug 15th, 2017, we taped out an ASIC implementation of an integrated chip using BOOM as its processing core in TSMC 28 nm HPM (high performance mobile). We call the chip *BROOM*, an acronym for the *Berkeley Resilient Out-of-Order Machine*. Figure 3.3 shows the block-diagram of the *BROOM* processor. *BROOM* consists of a single BOOM core and a 1 MB L2 cache, each in their own clock (and voltage) domains.

The main focus of the research chip is to study SRAM resiliency techniques. The techniques being explored allow the chip to lower the voltage on its SRAMs to improve the energy efficiency of the processor. This work builds on prior resiliency efforts [54, 127].

BROOM was implemented using LVT-based standard cells and a foundry-provided memory compiler. The entire chip measures 2 mm by 3 mm and is composed of 72 million transistors. The chip is composed of 417k standard cells and 73 SRAM macros; the core and L1 caches make up 310k cells and 20 SRAM macros. The final sign-off in the `slow-slow` corner was at 1.55 ns. Figures 3.4 and 3.5 show the place-and-routed chip plot (with and without annotations).

BROOM uses 124 external IO pads and is wire-bonded to a PCB using chip-on-board packaging, as shown in Figure 3.6. A serial interface provides communications to a Xilinx ZedBoard Zynq-7000 FPGA which provides the glue logic between *BROOM*, a host computer, and *BROOM*'s off-chip DRAM (mounted on the FPGA).

We successfully demonstrated booting Linux 4.6.2 on *BROOM* silicon on Dec 12, 2017.

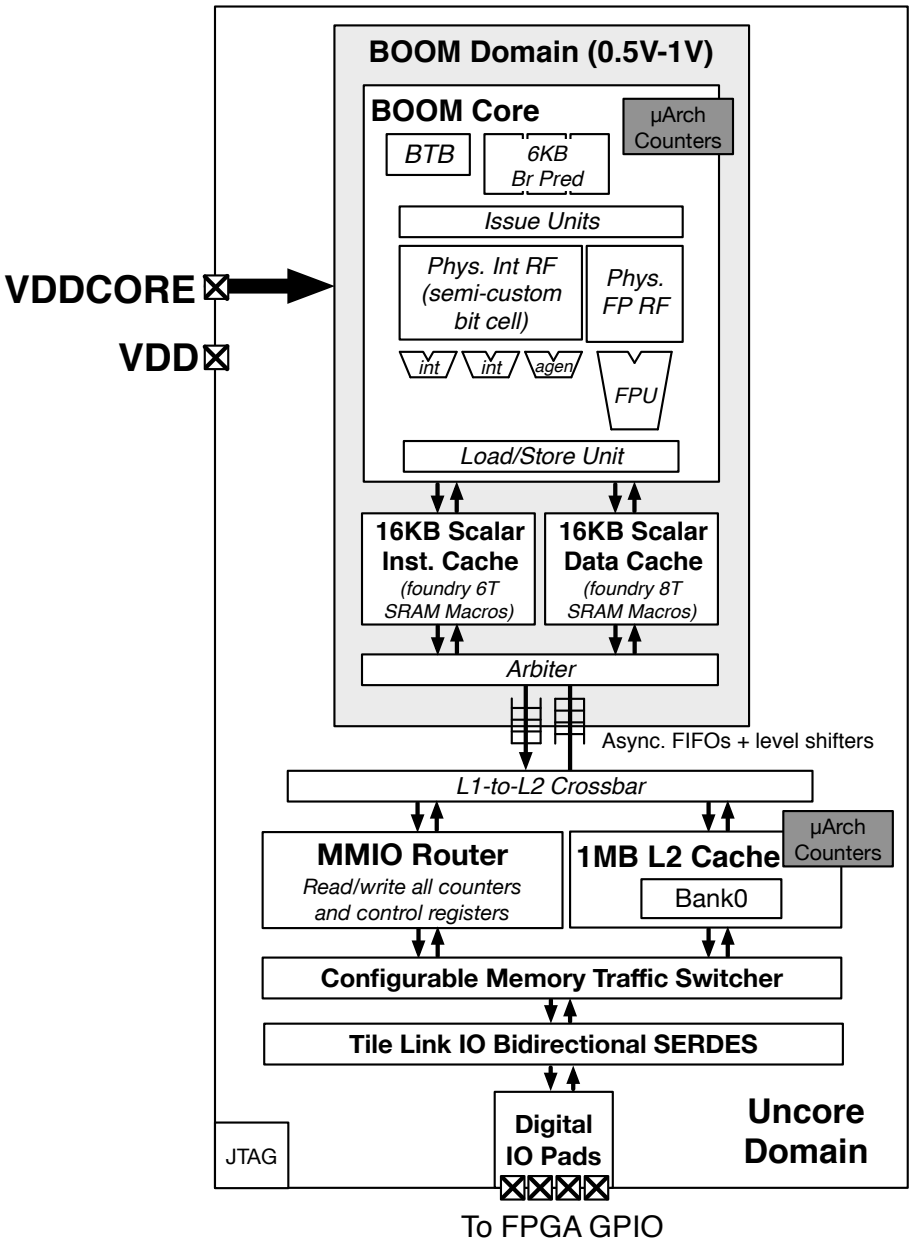


Figure 3.3: Block diagram of the BROOM test chip.

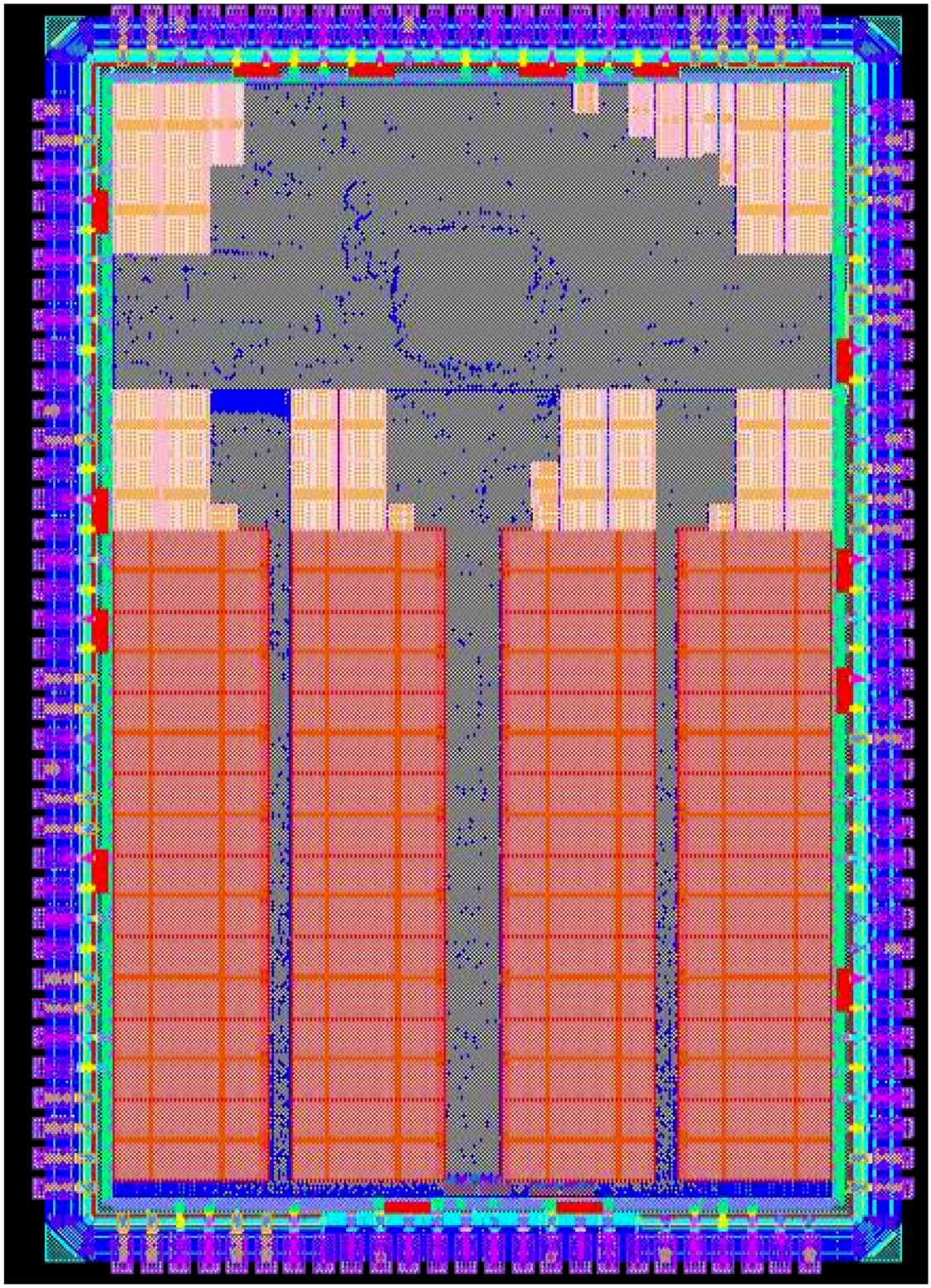


Figure 3.4: BROOM place-and-routed chip plot.

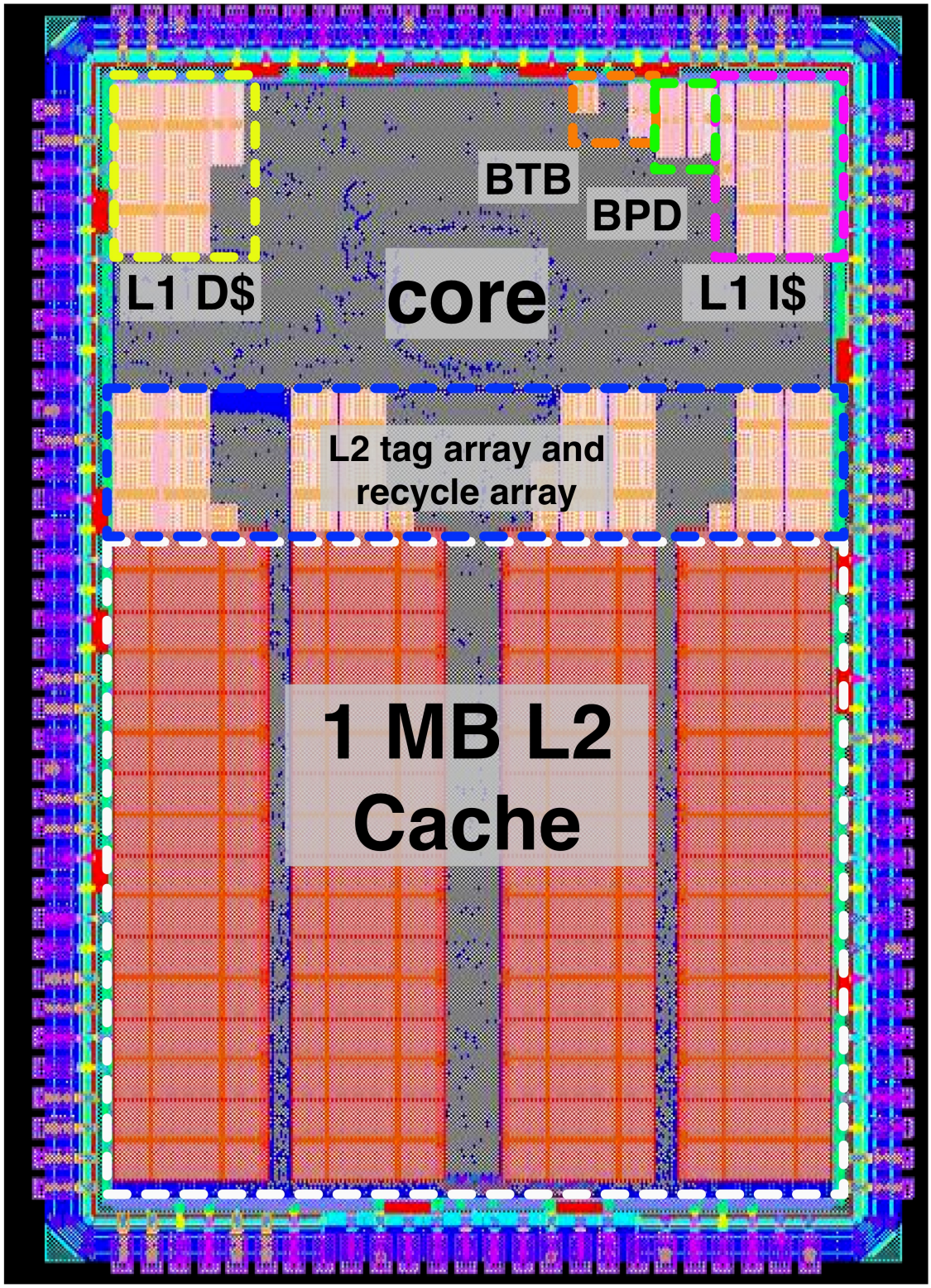


Figure 3.5: BROOM place-and-routed chip plot with annotations.

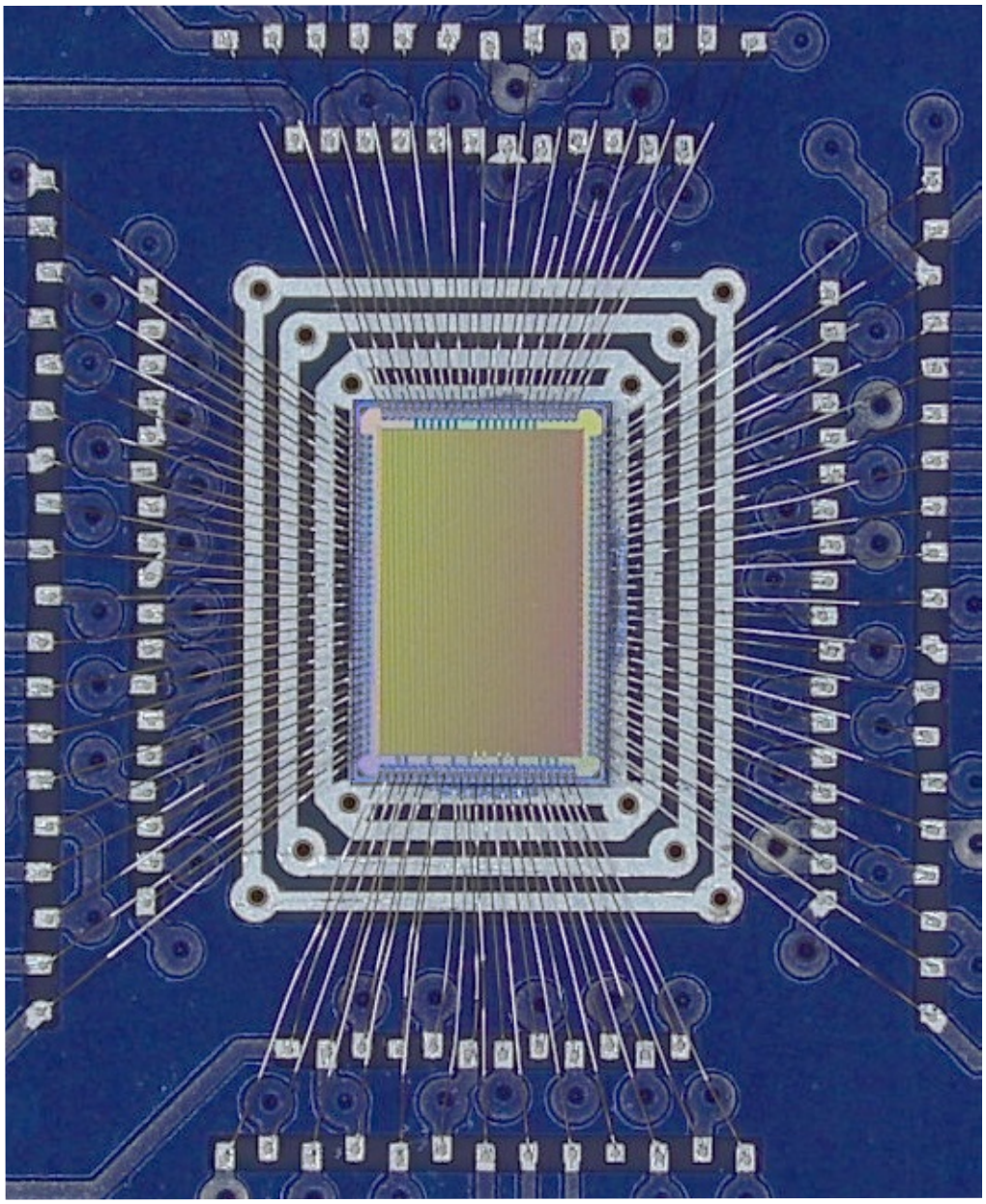


Figure 3.6: Photograph of a BROOM chip wire-bonded into a PCB via chip-on-board packaging.

3.5.1 SRAM Generation

SRAM memories are used to implement the relatively dense data arrays for the L1 instruction cache, the L1 and L2 data caches, the conditional branch predictor, and the branch target buffer.

For general logic, synthesizable Verilog RTL is generated from the *Chisel* RTL description and run through the synthesis, place, and route tools. For targeting SRAM memories, *Chisel* provides a `SeqMem` construct. The `SeqMem` construct provides a memory array with a synchronous read semantics and undefined behavior for reads that occur simultaneous with a write. Chisel can either provide its own Verilog behavioral model for `SeqMem`s for RTL simulation or flip-flop-based synthesis, or it can blackbox any `SeqMem` instantiations that use one or two ports. For every `SeqMem` that is blackboxed by *Chisel*, an entry is added to a *Chisel*-generated configuration file that lists the parameters of each `SeqMem`, including the number and type of ports, the width and depth of the memory, and the bit-mask behavior of the write port. The designer can then provide an SRAM instance in place of the blackboxed `SeqMem` description that matches its parameters.

Table 3.3 shows the configurations for each of the SRAMs used within a BOOM core as chosen for the *BROOM* tapeout. Some of the memories require masked writes to allow for partial updates to an entry. The L1 data cache is the only SRAM in the core that requires two ports. Two ports allows the for error correction schemes to read, modify, and write (correct) the tags or data arrays while maintaining full request throughput.

For the *BROOM* tapeout, the conditional branch predictor (BPD) used is a `gshare` predictor. BOOM's `gshare` predictor tracks the last 13 branch outcomes and hashes this "global history" with the lookup address to index into an array of two-bit counters. As a particular branch is *taken*, the counter saturates upwards; and if *not taken*, the counter saturates downwards. To fit the two-bit counters into single-port SRAM, the two-bit counters are physically separated into a *prediction (p) table* and a *hysteresis (h) table*. The p-table is further banked to allow for reads (predictions) and writes (updates) to occur simultaneously. A single *h-bit* is then shared across two *p-bit* entries. The resulting `gshare` structure is three equal-size single-port SRAMs. Section 4.2 discusses the BPD in more detail.

We used a TSMC-provided memory compiler to generate the SRAM memories. A `Python` script generated the SRAM instantiation code by analyzing the *Chisel*-generated SRAM configuration file and choosing the appropriate memory from a list of available foundry-provided memories.

3.5.2 Custom Bit Array Register File

The BOOM integer register file used in the *BROOM* tapeout is a 6-read, 3-write register file with seventy 64-bit registers. We chose to implement the register file by manually crafting a register file *bit* out of foundry-provided standard cells. We then pre-placed each register bit in an array and let the placer automatically route the wires to and from each bit. We

Table 3.3: The configurations used for each of the SRAMs used in a BOOM core. The number of uses, or instances, is also shown. For example, the L1 data cache is set-associative with four ways.

Use	Depth (entries)	Width (bits)	Size (bytes)	Ports	Mask Granularity	Instances
L1 D$ tags	64	88	704	w+r	22	4
L1 D$ data	256	128	4096	w+r	64	4
L1 I$ tags	64	80	640	rw	20	4
L1 I$ data	256	128	4096	rw	-	4
BTB tags	64	20	160	rw	-	2
BTB data	64	44	352	rw	-	2
BPD counters	128	64	1024	rw	1	3

further divided up the bits into clusters for hierarchical read lines; tri-states drive the read ports inside of each cluster and muxes select the read data across clusters.

The bits were described in structural Verilog and manually instantiated the foundry-provided standard cells. We implemented the decode logic in Chisel and then blackboxed the register file. We implemented a behavioral model of the custom array in Chisel to verify the decode logic through RTL simulation and then performed additional verification of the custom bit-array register file in gate-level simulation.

Aside from the integer register file and the SRAMs, no other logic in *Chisel* was implemented via blackboxes.

3.5.3 Timing Analysis

Timing analysis was performed by analyzing the timing reports generated from Synposys Design Compiler (synthesis) and Synposys IC Compiler (place-and-route) while targeting the `slow-slow` corner.

Most critical paths required a small amount of refactoring to remove late-arriving signals off the critical path. Other critical paths could be quickly solved by moving logic that is "out-of-band from the main instruction pipeline" to an extra stage. One example is the updating of the Branch Target Buffer (BTB) from a mispredicted branch as detected by the Branch Resolution Unit (BRU) in the backend. Although the instruction fetch redirect should happen as soon as possible, updating the BTB to correctly predict the branch on its next execution is less critical.

An other helpful tool with addressing critical paths is *register retiming*. Register retiming allows the tools to move the exact location of a register to improve timing results without changing the functionality of the circuit. The ability of the VLSI tools to utilize register retiming is repeatedly leveraged throughout the BOOM and *Rocket-chip* code base to reduce the code complexity and improve quality-of-result. We directed the tools to use register retiming on the following modules:

- Arithmetic Logic Unit (ALUUnit)
- Pipelined Integer Multiplier (Imul)
- Fused-Multiply-Add Pipeline (FPUFMAPipe)
- Integer-to-FP Conversion Unit (IntToFP)
- FP-to-FP Conversion Unit (FPToFP)
- Rename Stage (RenameStage – flattened and retimed)
- Branch Prediction Pipeline (BranchPredictionStage)
- Fetch Unit (FetchUnit)

3.6 Parameters

Table 3.4 shows the parameters chosen for the BOOM tapeout. Our overall strategy was to begin with a modestly sized processor, find and improve its critical paths, and then increase the data structures (such as the branch predictor tables) to match the available area and timing slack. Unfortunately, we found ourselves mostly limited in area. In response, we reduced the L1 caches from 32 kB to 16 kB.

3.7 Performance Evaluation

Table 3.5 shows results of area, frequency, and performance comparisons of BOOM against other industry processors using the CoreMark EEMBC benchmark. Our aim is to be competitive in both performance and area against low-power, embedded out-of-order cores. Bolded is the BOOM configuration taped out as part of the *BROOM* project using the parameters listed in Table 3.4, labeled BOOMv2. Two older variants of BOOM, labeled *BOOMv1 two-wide* and *BOOMv1 four-wide*, demonstrate the high-water mark of BOOM when fetching two and four instructions per cycle respectively.

BOOMv1 enjoys higher performance over BOOMv2 largely for two reasons. First, BOOMv1 uses the more complex, but more accurate TAGE branch predictor [89]. Second, BOOMv2 adds an extra cycle of latency to the load-use delay.

These performance loss is not fundamental to the design of BOOMv2. We chose to minimize design risk of the *BROOM* tapeout by using a simpler `gshare`-based branch predictor. The load-use delay can be addressed by adding a load-to-use bypass, cutting 5 cycles of load-use delay to 3 cycles.

BOOMv1 was designed with only limited access to educational technology libraries and no access to a memory compiler, and as such offers a shorter pipeline with more complexity per stage. In contrast, BOOMv2 was designed with feedback provided by the TSMC 28 nm HPM libraries and a foundry-provided memory compiler. Chapter 6 covers the design decisions made in progressing from BOOMv1 to BOOMv2.

Figures 3.7 and 3.8 show performance comparisons while running 8 benchmarks from the SPECint2006 benchmark suite. All data was taken by running each benchmark to completion — roughly 12 trillion instructions per processor. Figure 3.7 shows the performance data normalized to clock frequency. Rocket and BOOM data was collected on an FPGA using a memory model that accurately simulates a 1 MB L2 cache with a 20-cycle access latency and off-chip memory with a fixed 80-cycle access latency.

Figure 3.8 shows the full run-time performance by normalizing the target run time (in seconds) to the SPEC 2006 reference machine, a 296 MHz UltraSPARC II. BOOM's overall simulated performance is on par with the out-of-order ARM Cortex-A9. As the BOOM and Rocket performances are captured using an FPGA, the frequency used to calculate target run time in seconds was chosen to match the frequency of the Cortex-A9 and is not based on any particular silicon chip. Neither *BROOM* nor the other available silicon Rocket research

Table 3.4: The parameters chosen for the tapeout of BOOM. Although BOOM is a parameterizable generator, we must commit to a single configuration for a silicon instantiation.

Fetch Width	2 instructions
Issue Width	4 micro-ops (2 int, 1 mem, 1 fp)
Issue entries (int) Issue entries (mem) Issue entries (fp)	16 20 10
ROB entries Load/Store Unit	48 16 loads / 16 stores
FP Div/Sqrt	disabled
Branch Target Buffer Return Address Stack Conditional Branch Predictor	64 sets x 2 ways 8 entries 13 bits of global history (gshare)
L1 Cache (instruction)	64 sets x 4 ways (16 kB)
L1 Cache (data)	64 sets x 4 ways (16 kB)
Perf Counters Perf Events	6 counters 37 events
Regfile	6r3w (int), 3r2w (fp)
Exe Units	iALU+iMul+iDiv iALU FMA+fDiv Load/Store
FPU latency	4 cycles (SFMA, DFMA)

Table 3.5: CoreMark, area, and frequency comparisons of industry processors. The BOOMv1 designs enjoy higher performance relative to the BOOMv2 chip due to its more sophisticated TAGE branch predictor and its shorter load-use delay. Idealistic, perfect scaling is assumed to provide a normalized area measurement across technologies.

Processor	Core Area (core+L1s)	Scaled Area (to 28 nm)	CoreMark/ MHz/Core	Freq (MHz)		CoreMark/ Core		IPC
Intel Xeon E5 2687W (Sandy)†	≈18 mm^2 @ 32nm	14 mm^2	**7.36**	3,400		25,007		-
Intel Xeon E5 2667 (Ivy)*	≈12 mm^2 @ 22nm	19 mm^2	**5.60**	3,300		18,474		1.96
RV64 BOOMv1 four-wide	1.4 mm^2 @ 45nm	0.54 mm^2	**5.18**	n/a	**	n/a	**	1.50
RV64 BOOMv1 two-wide	1.1 mm^2 @ 45nm	0.43 mm^2	**4.87**	n/a	**	n/a	**	1.25
ARM Cortex-A15*	2.8 mm^2 @ 28nm	2.8 mm^2	**4.72**	2,116		9,977		1.50
RV64 BOOMv2 (BROOM chip)	0.52 mm^2 @ 28nm	0.52 mm^2	**3.77**	1,250	§	4,713	§	1.11
ARM Cortex-A9 (Kayla Tegra 3)*	≈2.5 mm^2 @ 40nm	1.2 mm^2	**3.71**	1,400		5,189		1.19
MIPS 74K‡	2.5 mm^2 @ 65nm	0.46 mm^2	**2.50**	1,600		4,000		-
RV64 Rocket*	0.5 mm^2 @ 45nm	0.19 mm^2	**2.32**	1,500	**	3,480	**	0.76
ARM Cortex-A5‡	0.5 mm^2 @ 40nm	0.25 mm^2	**2.13**	1,000		2,125		-

Results collected from *the author (using `gcc51 -O3` and `perf`), †[26], or ‡ [6]. The Intel core areas include the L1 and L2 caches.

** Results collected from RTL simulations. Area data provided via Cacti [122] and educational VLSI libraries. Frequency data not available as designs were not fully pushed through to a DRC clean tape-in.

§ Frequency based off of place-and-route data.

chips include a DDR PHY, making it unrealistic to run SPEC workloads. Without a DDR PHY, the off-chip serial interface is the weakest link in running SPEC 2006 due to its 2 GB memory requirement.

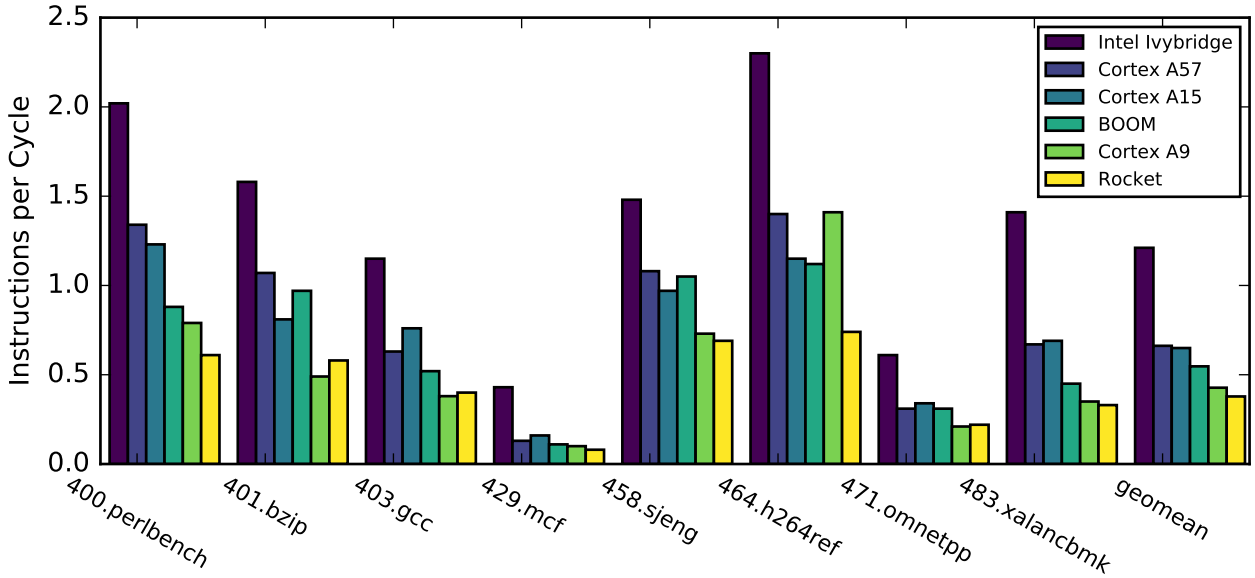


Figure 3.7: Instruction-per-cycle comparison running SPECint2006. BOOM and Rocket performance is measured by running each benchmark to full completion using `reference` input sizes on an FPGA with a simulated memory system. The values plotted here are also shown in Table 3.6.

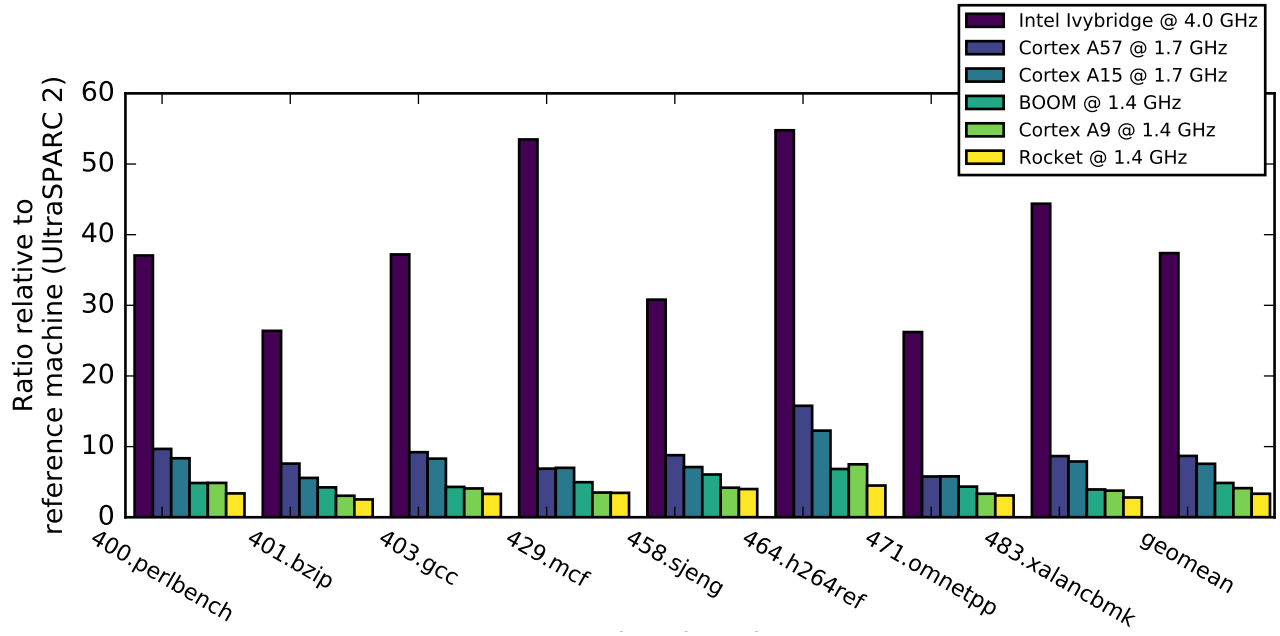


Figure 3.8: Performance ratio relative to the SPEC reference machine (a 296 MHz UltraSPARC II). BOOM and Rocket performance is measured by running each benchmark to full completion using `reference` input sizes on an FPGA with a simulated memory system. The frequency of BOOM and Rocket was chosen to match the frequency of the Cortex-A9. The values plotted here are also shown in Table 3.7.

Table 3.6: Instruction-per-cycle comparison running SPECint2006. BOOM and Rocket performance is measured by running each benchmark to full completion using `reference` input sizes on an FPGA with a simulated memory system.

benchmark	Intel Ivybridge	Cortex-A57	Cortex-A15	BOOM	Cortex-A9	Rocket
400.perlbench	2.02	1.34	1.23	0.88	0.79	0.61
401.bzip	1.58	1.07	0.81	0.97	0.49	0.58
403.gcc	1.15	0.63	0.76	0.52	0.38	0.40
429.mcf	0.43	0.13	0.16	0.11	0.10	0.08
458.sjeng	1.48	1.08	0.97	1.05	0.73	0.69
464.h264ref	2.30	1.40	1.15	1.12	1.41	0.74
471.omnetpp	0.61	0.31	0.34	0.31	0.21	0.22
483.xalancbmk	1.41	0.67	0.69	0.45	0.35	0.33
geomean	1.21	0.66	0.65	0.55	0.43	0.38

Table 3.7: Performance ratio relative to the SPEC reference machine (a 296 MHz UltraSPARC II). BOOM and Rocket performance is measured by running each benchmark to full completion using `reference` input sizes on an FPGA with a simulated memory system. The frequency of BOOM and Rocket was chosen to match the frequency of the Cortex-A9.

benchmark	Intel Ivybridge	Cortex-A57	Cortex-A15	BOOM	Cortex-A9	Rocket
400.perlbench	37.1	9.7	8.4	4.9	4.9	3.4
401.bzip	26.4	7.6	5.6	4.2	3.0	2.5
403.gcc	37.2	9.2	8.3	4.3	4.1	3.3
429.mcf	53.5	6.9	7.0	5.0	3.5	3.5
458.sjeng	30.8	8.8	7.1	6.1	4.2	4.0
464.h264ref	54.8	15.8	12.3	6.8	7.5	4.5
471.omnetpp	26.2	5.8	5.8	4.3	3.3	3.1
483.xalancbmk	44.4	8.7	7.9	3.9	3.8	2.8
geomean	37.4	8.7	7.6	4.9	4.1	3.3

Chapter 4

Branch Prediction

Branch prediction is a crucial component to the performance of advanced processors. Normally, branch instructions interrupt the stream of instructions as a branch that is *taken* redirects the frontend to begin fetching instructions from a new memory location. As the branch *direction* cannot be resolved until much later in the processor's execution pipeline, an unacceptable delay is encountered between branch *resolution* and fetching the subsequent instructions that come after the branch. A common solution to this problem is to use *speculation* — predict the direction of the branch, make forward progress under the assumption the prediction was correct, and rollback any changes if a *misprediction* occurred. By predicting the direction of branch instructions, the processor can continue making forward progress with no expensive delays. Instead, a penalty is paid only when the branch prediction is incorrect. Thus, processor performance is best when the instruction fetch unit (the frontend) is able to provide an uninterrupted stream of instructions to the execution backend.

Any branch mispredictions in the frontend will not be discovered until the branch instruction is executed later in the backend, typically 10s to 100s of cycles later. In the event of a misprediction, all instructions after the branch must be flushed from the processor and the frontend must be restarted using the correct instruction path. As modern processors tend to support 100-200 instructions inflight, a branch misprediction can be a significant performance hit. For example, if one in ten instructions is a branch, and a branch predictor is 90% accurate, then 1% of instructions will be a mispredicted branch. If the average misprediction penalty is 20 cycles, then for every 100 instructions the pipeline suffers a 20 cycle penalty. For a workload that is otherwise enjoying an instruction / cycle (IPC) throughput of 1.0, the 90% branch prediction accuracy translates to a 20% performance slowdown. Even if accuracy is increased to 95%, the pipeline is still suffering a 10% slowdown.

This chapter discusses the challenges in mapping branch prediction strategies studied using software simulators to synthesizable hardware implementations. First, Section 4.1 briefly discusses the state-of-the-art in branch prediction research and the challenges in mapping these research software implementations to synthesizable hardware implementations. Section 4.2 then describes BOOM's implementation of a branch predictor in *Chisel* and how we translated the implementation described by [89] (and related papers) to an RTL description

for use by BOOM. Section 4.3 discusses how we explicitly addressed the gaps between RTL and software models. This chapter concludes in Section 4.4 with some proposed improvements to the existing software model frameworks for better capturing the complexity and costs of branch predictors.

4.1 Background

This section provides some background and motivation to better understand the context of BOOM's branch prediction framework. First, this section discusses the research methodology used to evaluate the current state-of-the-art predictor designs. Next, this section explores the gaps between the current research methodology and the issues that arise in designing RTL implementations of branch predictors. Finally, this section provides a high-level overview of the state-of-the-art TAGE branch predictor [89].

4.1.1 Research Methodology

Branch prediction has been a popular focus of research for over three decades [97, 65]. Branch prediction is critically important to a processor's performance, but part of this longevity is in part due to the ease at which branch prediction strategies can be evaluated and explored. Prediction strategies can be evaluated by feeding predictor models a sequence of branch instructions from an instruction trace one at a time. At each branch, the predictor model is given the *fetch address*. The predictor model then provides its prediction — *taken* or *not-taken*. Once the prediction has been performed, the simulation framework tells the branch predictor model that correct outcome and the predictor can update itself as appropriate. These *un-pipelined, one-at-a-time* models are simple and usually much less than a 1,000 lines of C++ code. No other part of the processor needs to be implemented. The branch predictor implementations need only provide predictions, and any correctness bugs in the predictor have no impact on the functional behavior of the program traces. Also, the simplicity of these trace-based simulators allow branch prediction models to simulate at a rate of roughly 10 million instructions per second (MIPS), far faster than other, full-featured processor simulators.

One example framework of this method is the Championship Branch Prediction Contest (CBP) [21]. The CBP is a contest held roughly every two years and co-located at the International Symposium of Computer Architecture (ISCA) conference, whose organizers provide the CBP simulator infrastructure and simulation traces. In the last CBP Contest, all implementations were perceptron-based designs [50] or TAGE-based designs [89] (with one design being a combination of both [49]). Both predictor styles have been reported to be used by industry [18, 125] although specific implementation details or design deviations from the academic versions have not been disclosed.

4.1.2 Deficiencies of Trace-based, Unpipelined Models

Trace-based simulation has proven very useful for quickly prototyping branch prediction strategies and discovering an upper-bound on the accuracy for a particular implementation. However, it fails to capture some of the complexity of a superscalar processor that makes it challenging to translate predictor models to RTL implementations [31]. These gaps arise both from the trace-based nature of the model and from the sequential one-at-a-time, unpipelined aspect of the model. Only committed instructions are executed, only one instruction is predicted at a time, and that one instruction is then immediately updated before predicting the next instruction.

The first gap between models and RTL implementations is that the models only predict on a single branch instruction at a time. In comparison, high performance processors may fetch around 4-8 instructions (or more) a cycle. The more instructions fetched in a cycle the more likely the processor will fetch multiple branches in the same *fetch packet*. There are a few ways to deal with superscalar prediction — one could perform 8 predictions in parallel and choose the first taken branch [93], or one could perform only one prediction and track the branch instruction responsible for the prediction for a particular *fetch address*. Unfortunately, trace-based simulation provides no guidance in evaluating the tradeoffs between the two approaches. There is also the issue of global history — does it track the outcome of all branches in the program or does it track history only on the granularity of a *fetch cycle* ("was any branch taken within this *fetch packet*?").

Second, there is a delay between initiating a branch prediction and changing the flow of instructions based on the prediction. For BOOM this delay is 3 cycles: 1) compute table index by hashing the *fetch address* and the global history, 2) read the prediction tables, 3) and make the final decision and redirect the *fetch address*. This delay means that the global history used to hash into the predictor cannot completely contain all branches that have occurred before it in the program — there may be some branches hiding in this *prediction shadow*.

Third, there is a delay between performing the prediction in the *Fetch* stages and updating the predictor with the resolved outcome in the *Execute* or *Commit* stages. Some predictors may want to bypass inflight updates to new predictions, but the complexity of this bypass structure is not captured in the simpler un-pipelined, one-at-a-time models. These simple un-pipelined models also do not properly capture the complexity of storing the prediction state required to update the predictor in the *Execute* or *Commit* stages.

Fourth, designs are typically judged by the amount of state bits required to make predictions. This does not account for the additional state required to reset the predictor on a misprediction or exception, which for some designs can become very expensive. Some proposed designs track over a thousand bits of history, which must be snapshotted and reset on a misprediction. The winning entry in the CBP-4 4 kB space was TAGE-SC-L [91] with 10 tagged tables and a history length of 359 bits. Some of the implementation suggestions for TAGE recommend using three circular shift registers (CSR) of roughly 10-13 bits each per table to track indexing and tag computations. Each CSR must be snapshotted per pre-

diction — for the winning design, this comes to roughly 250 bits of CSR state per *fetch cycle*. In order to manage 40 cycles-worth of CSR state storage, the predictor would have to add at least 30% to its total state storage on top of its existing 4 kB state used for the prediction tables. Also, not all bits are created equal. Designs that can cram the prediction tables into single-port SRAM can enjoy greater density, whereas more complex predictors may require the use of multi-port flip-flop-based arrays which are much more expensive in area and power.

Finally, trace-based models do not capture the implementation complexity and realism. For example, the `2bc-gskew` predictor used in the (cancelled) Alpha EV8 processor purposefully avoids a *local history predictor* due to the number of read ports required by the additional level of indirection provided by the local histories [93]. Instead, trace-based modeling requires care from the model builder to stay within the realm of possibility [90].

4.1.3 The State-of-the-art TAGE Predictor

TAGE predictors are regarded as the most accurate of academic implementations, as judged by the CBP contests, and can be found in processor simulators such as ESESC [5].

TAGE is composed of a small number of *tables*. As shown in Figure 4.1, each table entry consists of a *tag*, an n-bit saturating *prediction* counter, and a saturating *usefulness* counter. TAGE begins making its prediction by hashing the *fetch address* and the *global history* to generate an index and a tag for each table. Each table uses geometrically more history than the previous table as part of its hashing function. For example, the first table may use only 5 bits, the second table 17 bits, the third table 44 bits, and the fourth table 130 bits. If the accessed entry has a tag hit, the table asserts a valid prediction and uses the prediction counter to provide the *taken/not-taken* prediction. The usefulness information can be used to decide which table should provide the prediction. Otherwise, the table with the longest history that experienced a tag hit makes the prediction.

When TAGE is updated on a misprediction, a new entry is allocated in one of the TAGE tables. If Table i made a prediction, then a new entry is allocated in some Table j such that $j > i$ but only if the current entry in Table j has a usefulness value of 0. If an allocation fails because an available entry cannot be found, the usefulness counter is decremented for each candidate entry for every table, making room for when the next allocation occurs.

A number of different techniques have been explored for degrading the usefulness counters to prevent the tables from filling up with rarely used entries that prevent new allocations from succeeding. These include clearing the bits after a fixed period of time or tracking how many allocation failures occur before degrading all of the usefulness counters.

There are four main insights that make TAGE particularly accurate compared to previous branch predictor designs.

First, TAGE adds tags to each entry. The heart of the TAGE predictor is the two-bit (or three-bit) saturating prediction counter. But attached to each counter is a tag to provide high confidence that a prediction from this counter truly matches with the *fetch address* and history. These tags are roughly 10-13 bits, roughly four or five times more expensive

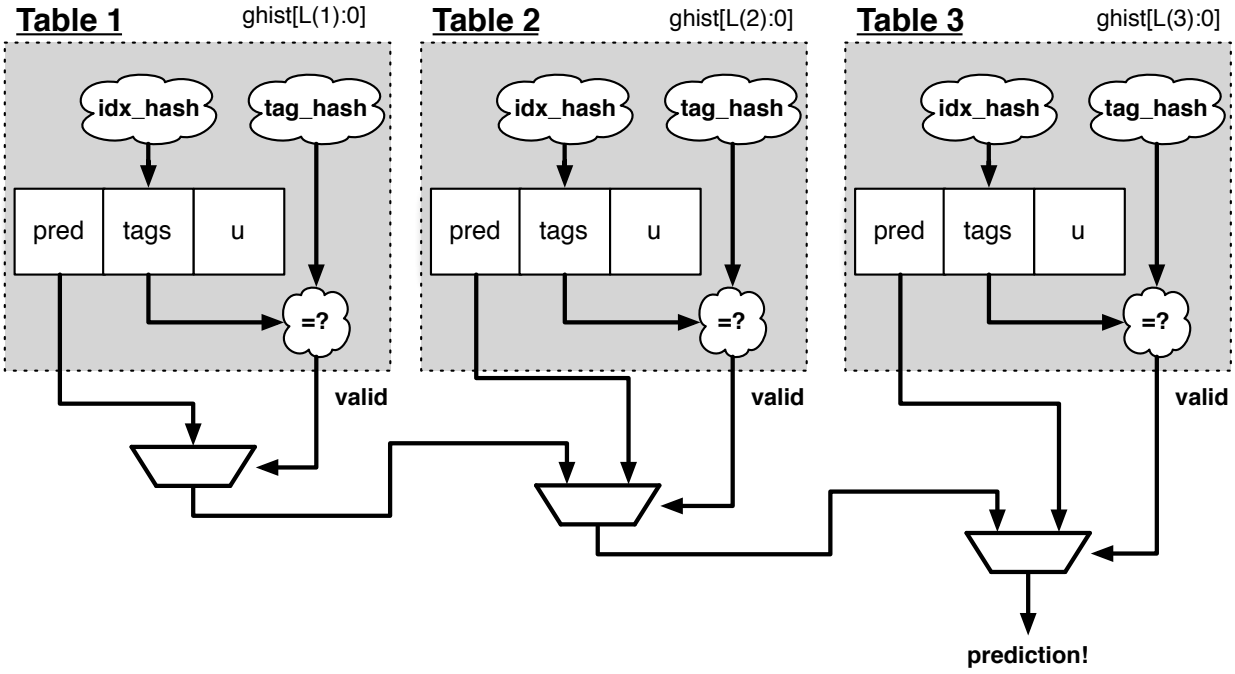


Figure 4.1: The TAGE predictor. The requesting address and the global history are fed into each table's index hash and tag hash functions. Each table provides its own prediction (or no prediction) and the table with the longest history wins. Each table has geometrically more history bits than the previous table.

than the prediction counters. Although intuition may suggest that a predictor with four or five times as many counters would address most aliasing concerns, the use of tags nearly completely eliminates aliasing and protects long-lived entries from being overwritten by sporadic branches.

Second, TAGE is able to exploit very long histories — hundreds of bits — by performing an XOR folding of the history to generate the index and tag hashes for each table. This provides much more reach than predictors than are limited to n bits of history for a predictor table of size 2^n.

Third, different branches require different lengths of history to provide the most accurate prediction. Some branches need very long histories to predict their direction while other branches need relatively short histories to prevent pollution from branches in unrelated program phases. Shorter histories also have the benefit of being faster to learn. Instead of having a single history length that must balance the needs of both types of branches, TAGE provides multiple prediction tables, each using a different history length. As each table uses a geometrically increasing amount of history to make its predictions, TAGE is able to accommodate both very long histories and very short histories.

Fourth, the usefulness counter provides a level of confidence behind each prediction — each table (and in turn TAGE itself) only provides a prediction when it is confident that the prediction is correct.

4.2 The BOOM RTL Implementation

This section discusses how BOOM predicts branches and then resolves these predictions, covering many of the implementation details of BOOM's branch prediction pipeline.

4.2.1 The Frontend Organization

The BOOM frontend covers instruction fetch and branch prediction as shown in Figure 4.2. Instruction fetch begins by sending a *fetch address* to the instruction cache. An entire cycle is devoted to accessing the SRAM within the instruction cache. In parallel to the instruction cache access, the Branch Target Buffer (BTB) is accessed. The BTB uses the *fetch address* to make a prediction on what the next cycle's *fetch address* should be. Each BTB entry contains a *tag* and a *target*. If the *fetch address* matches against a tag, then the target provides the prediction. The BTB is a very expensive structure, roughly 60 bits per entry, and it must make a prediction within a single cycle. Therefore, there is little room for clever algorithms. BOOM's BTB uses a bimodal predictor indexed by the *fetch address* to predict *taken/not-taken* on a tag hit. The BTB also tracks whether the stored control-flow instruction is a *return* or *call*, which interacts with a Return Address Stack predictor to predict function returns. Although the BTB is crucial to good processor performance, it is not the focus of this chapter. Instead, we focus on the larger conditional branch predictor,

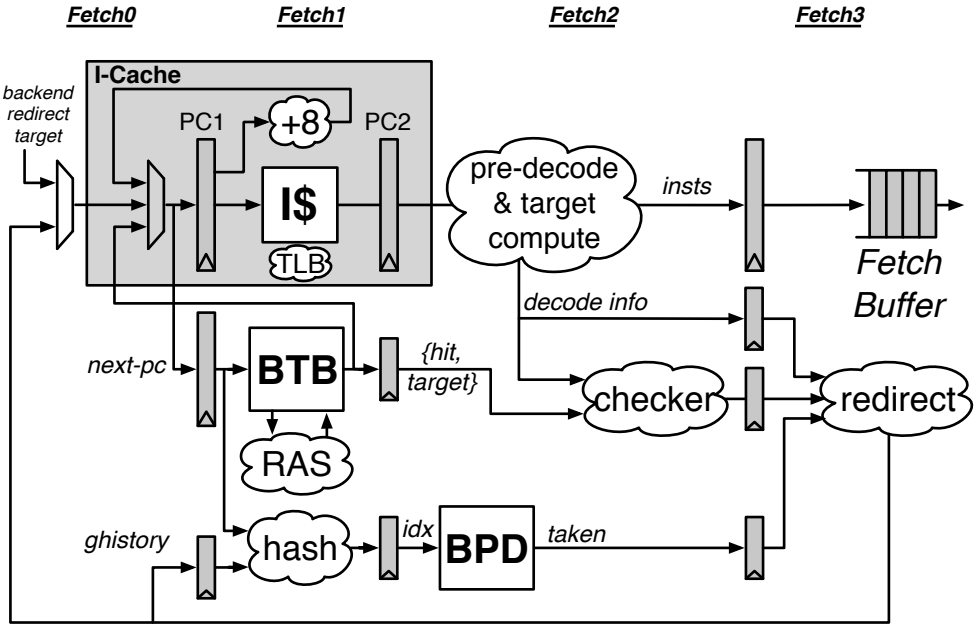


Figure 4.2: The BOOM Fetch Unit.

which relies on more sophisticated algorithms to accurately predict the *taken/not-taken* direction of conditional branches.

The conditional branch predictor is accessed via the *fetch address* and makes a set of *taken/not-taken* predictions a few cycles later. As BOOM's branch predictor takes 3 cycles to make a prediction, on a *taken* prediction BOOM's branch predictor squashes two stages of instruction fetch. As the conditional branch predictor only predicts *taken/not-taken*, it can store its predictions in very dense arrays, potentially spending as little as 1 or 2 bits per branch. For branch target information, the conditional branch predictor uses the instructions freshly fetched from the instruction cache to provide the branch targets.

In the *Fetch1* stage, while the BTB and instruction cache are being accessed, the conditional predictor first computes a hash from the *fetch address* and the *global history*. Conceptually, the global history (ghistory) tracks the outcome of the last n branches in the program. On the next cycle, *Fetch2*, this hash can then be used to index a particular set of entries within the branch predictor. In parallel with the prediction table access, the instructions returning from the instruction cache are searched for branches and the branch targets are computed. Lastly, in *Fetch3*, the predictor's predictions are matched against any branches that have been fetched, a final prediction is made, and the *fetch address* is redirected if the branch predictor disagrees with the current path set by the BTB. If a disagreement has occurred between the branch predictor and the BTB, 2 cycles of instruction fetch are squashed and the frontend starts fetching along a new branch path.

4.2.2 Providing a Branch Predictor Framework

Although there are a wide range of branch prediction algorithms, many fall into the same category of "global history predictors" in which the *fetch address* and the *global history* are used to make a prediction. Thus, it is possible to provide a branch predictor framework that facilitates implementing new designs by abstracting away most of the complexity that is shared across designs. In this manner, we can more easily explore different types of conditional branch predictors.

At a high-level, BOOM provides an abstract `BrPredictor` class. This class provides a standard interface into the branch predictor, it provides the control logic for managing the global history register, and it contains the *branch reorder buffer (BROB)* which handles the inflight branch prediction checkpoints. This abstract class can be found in Figure 4.3 labeled "predictor (base)".

4.2.3 Managing the Global History Register

The *global history register*, or *ghistory*, is an important piece of many branch predictor designs. It contains the outcomes of the previous N branches (where N is the size of the global history register). When fetching branch i, it is important that the direction of the previous $i - N$ branches is available so an accurate prediction can be made. Waiting till the *Commit* stage to update the global history register would be too late as dozens of branches

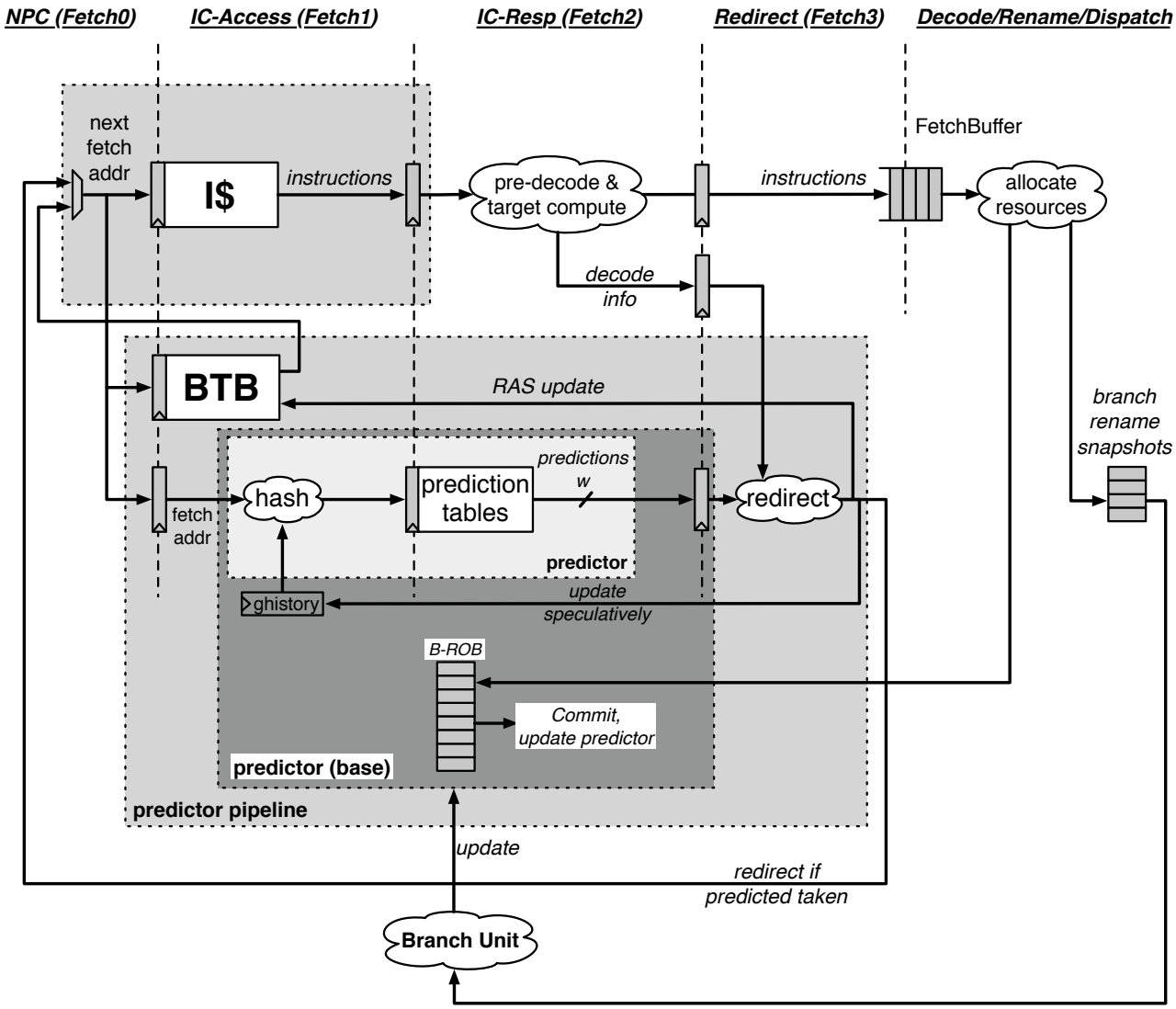


Figure 4.3: The branch prediction framework. The frontend sends the *next fetch address* to the branch prediction pipeline (*Fetch0* stage). A hash of the *fetch address* and the *global history* is used to index the predictor's prediction tables. The prediction tables are accessed in parallel with the instruction cache (*Fetch1* stage). The conditional branch predictor then returns a prediction for each instruction in the *fetch packet*. The instructions returning from the instruction cache are quickly decoded; any branches that are predicted as *taken* will redirect the frontend from the *Fetch3* stage. Prediction snapshots and metadata are stored in the *branch rename snapshots* (which are later used for fixing the predictor after mispredictions) and the *Branch Re-order Buffer (B-ROB)* (which is used to update the predictor in the *Commit* stage). The *predictor (base)* provides a generic, abstract implementation of a global history branch predictor. A new, concrete branch predictor implementation only needs to provide its hashing function(s) and prediction tables.

would be inflight and not reflected. Therefore, the global history register must be updated *speculatively*, once the branch is fetched and predicted.

Managing the global history faces three main challenges:

1. superscalar fetch and prediction
2. delayed update between fetch-initiation and redirection-due-to-prediction
3. resetting the global history on a misprediction or exception

To address superscalar fetch and prediction, the global history does not track individual branch outcomes, but rather compresses via an OR-reduction the direction of all conditional branches within the *fetch packet* into a single bit. As the superscalar fetch width increases, the odds of a *taken* branch increases. One solution is to hash in *path* history [73] to increase the randomness of the global history register [93].

If a misprediction occurs, the global history register must be reset and updated to reflect the actual history. This means that each branch (more accurately, each *fetch packet*) must snapshot the global history register in case of a misprediction. As previously mentioned, there is a delay between beginning to make a prediction in the *Fetch1* stage (when the global history is read) and redirecting the front-end in the *Fetch3* stage (when the global history is updated). This results in a "shadow" in which a branch beginning to make a prediction in *Fetch1* will not see the branches (or their outcomes) that came a cycle or two earlier in the program. These "shadow branches" must be reflected in the global history snapshot.

There is one final wrinkle - exceptional pipeline behavior. While each branch contains a snapshot of the global history register, any instruction can potential throw an exception that will cause a front-end redirect. Such an event will cause the global history register to become corrupted. For exceptions, this may seem acceptable - exceptions should be rare and the trap handlers will cause a pollution of the global history register anyways (from the point of view of the user code). However, some exceptional events include "pipeline replays" — events where an instruction causes a pipeline flush and the instruction is refetched and re-executed. For this reason, a *commit copy* of the global history register is also maintained by the abstract branch predictor class and is used for resetting the history on a pipeline flush event.

Some branch predictors such as TAGE require access to incredibly long histories – potentially over a thousand bits. Global history is speculatively updated after each prediction and must be snapshotted and reset if a misprediction was made. Snapshotting a thousand bits is untenable. Instead, the Very Long Global History Register (VLHR) is implemented as a circular buffer with a speculative head pointer and a commit head pointer. As a prediction is made, the prediction is written down at $VLHR[spec\_head]$ and the speculative head pointer is incremented and snapshotted. When a branch mispredicts, the head pointer is reset to $snapshot + 1$ and the correct direction is written to $VLHR[snapshot]$. In this manner, each

snapshot is on the order of 10 bits, not 1000 bits. BOOM provides two different implementations of ghistory — a shift register for short histories and the circular buffer for very long histories.

4.2.4 The Two-bit Counter Tables

The basic building block of most branch predictors is the "two-bit counter table" (2bc). As a particular branch is repeatedly taken, the counter saturates upwards to the max value 3 (`0b11`) or *strongly taken*. Likewise, repeatedly *not-taken* branches saturate towards zero (`0b00`). The high-order bit specifies the *prediction* and the low-order bit specifies the *hysteresis* (how strongly biased the prediction is).

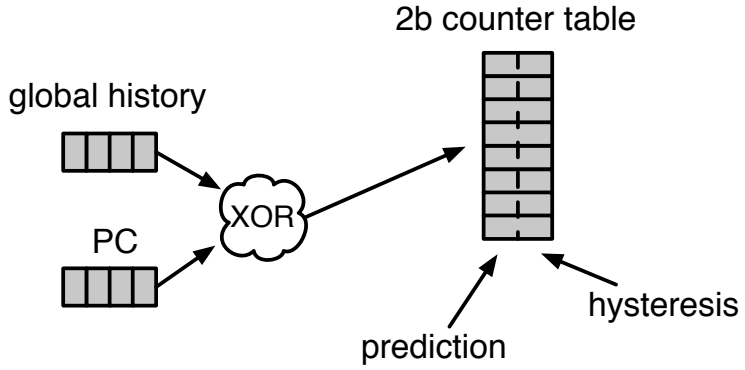


Figure 4.4: A *gshare* predictor uses the global history hashed with the *fetch address* (PC) to index into a table of 2-bit counters. The high-order bit makes the prediction.

These two-bit counters are aggregated into a table. Ideally, a good branch predictor knows which counter to index to make the best prediction. However, to fit these two-bit counters into dense SRAM, a change is made to the 2bc finite state machine – mispredictions made in the *weakly not-taken* state move the 2bc into the *strongly taken* state (and vice versa for *weakly taken* being mispredicted). The FSM behavior is shown in Figure 4.5.

Although it's no longer strictly a "counter", this change allows us to separate out the read and write requirements on the *prediction* and *hystersis* bits and place them in separate sequential memory tables. In hardware, the 2bc table can be implemented as follows:

The P-bit:

- **read** - every cycle to make a prediction
- **write** - only when a misprediction occurred (the value of the h-bit).

The H-bit:

- **read** - only when a misprediction occurred.

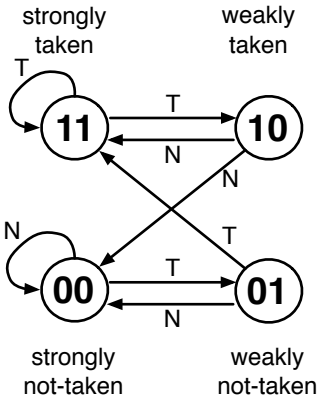


Figure 4.5: Two-bit counter state machine.

- **write** - when a branch is resolved (write the direction the branch took).

By breaking the high-order p-bit and the low-order h-bit apart, we can place each in a two-ported (1r1w) SRAM. By making a few assumptions we can reduce this to a single-ported (1rw) SRAM. We can assume that mispredictions are rare and branch resolutions are not necessarily occurring on every cycle. Also, writes can be delayed or even dropped altogether. Therefore, the *h-table* can be implemented using a single 1rw-ported SRAM by queueing writes up and draining them when a read is not being performed. Likewise, the *p-table* can be implemented using a 1rw-ported SRAM by banking the SRAM, buffering writes into a queue, and draining the queue when there is not a read conflict for that particular bank.

We also have to be aware that SRAMs have a particular ratio of depth to width where they provide the best physical characteristics. The "tall and skinny" aspect ratio that the 2bc tables require is a particularly bad design point. However, the solution is simple – tall and skinny can be trivially transformed into the preferred rectangular memory structure. The high-order bits of the index can correspond to the SRAM row and the low-order bits can be used to mux out the specific bits from within the row. We have created a memory utility class in *Chisel* that automatically transforms our single-ported memories into rectangular structures.

4.2.5 Superscalar Predictions

A superscalar fetch unit fetches multiple instructions every cycle.

Thus, the BTB and conditional branch predictor must predict across the entire *fetch packet* which of the many possible branches will be the dominating branch that redirects the *fetch address*. This raises a challenge for structures like the BTB, which uses tag lookups to facilitate their predictions — exactly what do we store as the tag?

To address this issue, when the BTB makes a prediction, it performs a tag match against the predicted branch's *fetch address*, and not the address of the branch itself. That is to say,

when a branch is mispredicted, it informs the branch prediction structures of both its own address (so we know which instruction is a branch) but also the address that was used by the frontend to fetch the branch. In this manner, on future executions through the program, the *fetch address* of the branch is used for the tag match, and the predictor can store an *offset* to mark which instruction in the *fetch packet* is the branch.

Using this design, each BTB entry corresponds to a single *fetch address*, but it is helping to predict across the entire *fetch packet*. The BTB entry can only store meta-data and target-data on a single control-flow instruction. While there are certainly pathological cases that can harm performance with this design, the assumption is that there is a correlation between which branch in a *fetch packet* is the dominating branch relative to the *fetch address*, and evaluations of this design using a trace-based superscalar fetch unit simulator has shown this design is very complexity-friendly with no noticeable loss in performance. A different design could instead choose to provide a whole bank of BTBs for each possible instruction in the *fetch packet* which would increase the number of entries accessed and add to the critical path to choose the winning *taken* branch.

4.2.6 The BOOM GShare Predictor

Gshare is a simple but very effective branch predictor. Predictions are made by hashing the instruction address and the global history (typically a simple XOR) and then indexing into a table of two-bit counters. Figure 4.4 shows the logical architecture and Figure 4.6 shows the physical implementation and structure of the *gshare* predictor. Note that the prediction begins in the *Fetch1* stage when the requesting address is sent to the predictor but that the prediction is made later in the *Fetch3* stage once the instructions have returned from the instruction cache and the prediction state has been read out of the *gshare*'s p-table. The heart of the *gshare* predictor is a two-bit counter table implemented as described in Section 4.2.4.

4.2.7 The BOOM TAGE Predictor

BOOM also implements the TAGE conditional branch predictor. TAGE is a highly-parameterizable, state-of-the-art global history predictor [89, 90, 92]. The design is able to maintain a high degree of accuracy while scaling from very small predictor sizes to very large predictor sizes. It is fast to learn short histories while also able to learn very, very long histories (over a thousand branches of history).

TAGE (TAgged GEometric) is implemented as a collection of predictor tables. Each table entry contains a *prediction* counter, a *usefulness* counter, and a *tag*. The *prediction* counter provides the prediction. The *usefulness* counter tracks how useful the particular entry has been in the past for providing correct predictions. The *tag* allows the table to only make a prediction if there is a tag match for the particular requesting instruction address and global history. More information about the TAGE design can be found in Section 4.1.3.

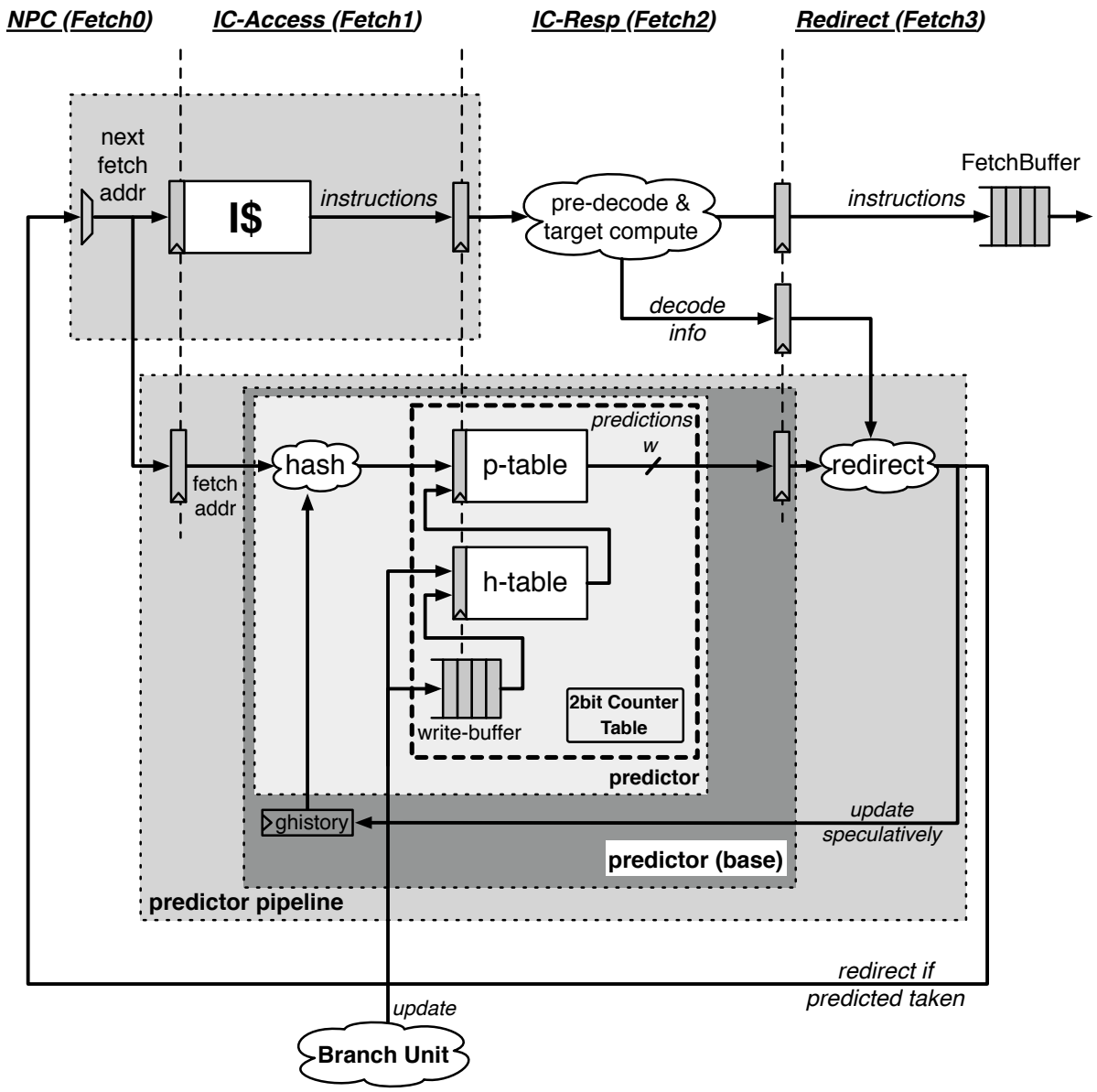


Figure 4.6: The *gshare* predictor pipeline. The two-bit counter table is split into a separate *p-table* (the high-order bit) and *h-table* (the low-order bit). Predictions only read the *p-table*. On every branch resolution, the Branch Unit writes the branch outcome to the *h-table*. If a misprediction occurs, the *h-table* is read and its value is written to the *p-table*.

4.2.7.1 TAGE Global History and the Circular Shift Registers (CSRs)

Each TAGE table has associated with it its own global history, and each table has geometrically more history than the previous table. As the histories become incredibly long (and thus too expensive to snapshot directly), TAGE uses the Very Long Global History Register (VLHR) as described in Section 4.2.3. The histories contain many more bits of history than can be used to index a TAGE table; therefore, the history must be "folded" to fit. A table with 1024 entries uses 10 bits to index the table. Therefore, if the table uses 20 bits of global history, the top 10 bits of history are XOR'ed against the bottom 10 bits of history.

Instead of attempting to dynamically fold a very long history register every cycle, each tage table stores its VLHR in a circular shift register (CSR). The history is stored already folded and only the new history bit and the oldest history bit need to be provided from the master copy of the VLHR to perform an update. Code 4.2.7.1 shows an example of how a CSR works.

```
1  Example:
2    A 12 bit value (0b_0111_1001_1111) folded onto a 5 bit CSR becomes
3    (0b_0_0010), which can be found by:
4
5
6                 /-- history[12] (evict bit)
7                 |
8   c[4], c[3], c[2], c[1], c[0]
9    |                          ^
10   |                          |
11   \________________________/ \---history[0] (newly taken bit)
12
13
14 (c[4] ^ h[ 0] generates the new c[0]).
15 (c[1] ^ h[12] generates the new c[2]).
```

Code 4.1: The circular shift register. When a new branch outcome is added, the register is shifted (and wrapped around). The new outcome is added and the oldest bit in the history is "evicted".

Each TAGE table must maintain three CSRs. The first CSR is used for computing the index hash and has a size $n = log(num\_table\_entries)$. As a CSR contains the folded history, any periodic history pattern matching the length of the CSR will XOR to all zeroes. This can potentially be a quite common occurrence. To deal with this issue, there are two CSRs for computing the tag hash, one of width n and the other of width $n - 1$.

For every prediction, all three CSRs (for every table) must be snapshotted and reset if a branch misprediction occurs. Another three *commit copies* of these CSRs must be maintained to handle pipeline flushes.

4.2.7.2 Usefulness Counters (u-bits)

The "usefulness" of an entry is stored in the *u-bit* counters. Roughly speaking, if an entry provides a correct prediction, the u-bit counter is incremented. If an entry provides an

incorrect prediction, the u-bit counter is decremented. When a misprediction occurs, TAGE attempts to allocate a new entry. To prevent overwriting a useful entry, it will only allocate an entry if the existing entry has a usefulness of zero. However, if an entry allocation fails because all of the potential entries are useful, then all of the potential entries are decremented to potentially make room for an allocation in the future.

To prevent TAGE from filling up with only useful but rarely-used entries, TAGE uses a scheme for "degrading" the u-bits over time. A number of schemes have been proposed. One option is a timer that periodically degrades the *u-bit* counters. Another option is to track the number of failed allocations (incrementing on a failed allocation and decremented on a successful allocation). Once the counter has saturated, all u-bits are degraded. BOOM implements the *u-bits* as a 1-bit counter stored in a flip-flop array, as they are otherwise too small to justify an SRAM and are easier to clear.

4.2.7.3 TAGE Snapshot State

For every prediction, all three CSRs (for every table) must be snapshotted and reset if a branch misprediction occurs. TAGE must also remember the index of each table that was checked for a prediction so the correct entry for each table can be updated later. TAGE must remember the tag computed for each table – the tags will be needed later if a new entry is to be allocated. Finally, commit copies of all CSRs are also maintained in the event of a pipeline exception.

4.3 Addressing the Gaps Between RTL and Models

In this section we discuss how the BOOM RTL implementation of a branch predictor has addressed the previously discussed deficiencies between the trace-based, unpipelined models and RTL.

4.3.1 Superscalar Prediction

Superscalar fetch brings a number of challenges to branch prediction. Although unpipelined simulators treat branch instructions as the basic quantum unit of execution, from the point-of-view of the RTL, the *fetch packet* is the basis on which predictions are made.

Given the current *fetch address*, the branch predictor must provide answers to the following questions:

1. What should the *fetch address* be for the next cycle?
2. If we predict *taken*, which branch in the *fetch packet* should take credit?

BOOM's BTB and conditional branch predictor take two different approaches to this problem.

The BTB performs a tag match using the *fetch address*. If there is a *tag hit*, the BTB provides an offset to denote which instruction in the *fetch packet* takes credit for the *taken/not-taken* prediction. This can be problematic if many branches exist within a *fetch packet* that alternate being *taken/not-taken*.

The conditional branch predictor instead uses a solution described by the EV8 processor [93]. A single lookup is performed based on an aligned *fetch address* which returns a bit vector of predictions. Each bit corresponds to a *taken/not-taken* prediction for each instruction in the *fetch packet*. This prediction vector can then be cross-referenced against the decoded *fetch packet* and the earliest predicted *taken* branch can be taken.

This method involves tracking the prediction counter state for all branches. In BOOM, we tracked the outcomes for all branches within each *fetch packet* and updated the branch predictor once the entire fetch packet had been committed. A write-mask allows BOOM to only update entries where an actual branch resides to reduce aliasing.

This method of tracking the prediction state of individual branches works well for global history predictors that do not use tags. However, for a given *fetch address* and global history, there is likely only a single branch that is *taken*. Therefore, for more complex predictors that add tags to each prediction entry, a more profitable solution is to only track the likely *taken* branch and its offset within the *fetch packet*.

Superscalar fetch also adds difficulty to the management of the global history register. Despite potentially fetching and predicting multiple branches every cycle, we reduce these outcomes to a single bit of history per cycle. To do this, we take inspiration from the EV8 [93], and reduce the *taken/not-taken* output across the entire *fetch-packet* via an OR-reduction.

4.3.2 Delayed History Update

The global history must reflect the outcome of all previous branches to the best of its ability. This means we must update the global history speculatively, as we make predictions. However, as there is a delay between initiating a branch prediction and resolving the prediction, the global history has a "shadow" in which branches are unable to see the direction of the branch in the most recent cycles.

For processors such as the Alpha EV8 with its 8-wide fetch, this delay is accepted as a "fact of life." However, the Alpha EV8 designers mention that the difference between their unpipelined simulator and their detailed Alpha EV8 micro-architectural simulator was "insignificant" and the 3-cycle delayed history only "slightly degrades" accuracy with a "limited" impact [93]. They found adding in some path history recovers most of the performance degradation. And for narrower-width processors, the odds of a branch being in the shadow are rarer.

Part of the challenge in dealing with the delayed history update was in resetting the history on a misprediction. On a misprediction on branch i, the global history must be reset for use by branch $i+1$ and updated to reflect the true outcome of branch i. This updated history must include the branches that were in the shadow (e.g., branches $i-1$ and $i-2$) so that branch $i+1$ sees the history it would normally see had branch i been correctly predicted.

This snapshot restoration of the global history gets even more difficult when a *fetch packet* contains multiple branches. If branch i at address a is mispredicted and its resolved direction is *not-taken*, the front-end is restarted at address $a + 4$[1]. If address $a + 4$ falls within the same *fetch packet* as address a, there may be another branch $i + 1$. In order to provide an accurate prediction of branch $i+1$, it must be provided a global history that does *not* include branch i. This is to prevent double counting of the *fetch packet* that includes branches i and $i + 1$, since, as discussed in the previous section, the entire outcome of the *fetch packet* is reduced to a single bit of history.

4.3.3 Delayed Predictor Update

BOOM performs its updates of the conditional branch predictor after commit. This can lead to some inaccuracy as a tight loop may have multiple iterations inflight and thus may continue to mispredict the branches within the loop before the first iteration can update the predictor. One solution is to bypass uncommitted updates to the fetch unit in a manner similar to a store buffer bypassing data to dependent loads [90].

4.3.4 Accurate Cost Measurements

Unpipelined models judge branch predictors on two metrics: the number of branch mispredictions and the number of bits used to track the prediction state.

However, not all branches have an equal effect on performance. One mispredicted branch may be completely hidden behind a cache miss while another branch misprediction may depend on a long latency operation and cause a large amount of wasted work to be discarded. Although not explored in this thesis, a detailed processor and branch predictor simulation may allow for more interesting ideas to be explored, such as detecting which branches are "critical" and allocating different resources to critical branches [34].

The other aspect more accurately captured by RTL is the area and energy costs of the branch predictor as opposed to the raw memory size of the predictor itself. First, some prediction structures more readily map to size-efficient single-ported SRAMs. Second, RTL implementations must be able to handle snapshot and recovery of the branch prediction state. Some predictor designs require a significant amount of snapshot state that is not tracked as a measurement in some branch prediction contests. For example, the gshare predictor only needs to track ≈15 bits of history for each prediction. However, one of the submitted TAGE predictor entries to the "4kB limited size" branch predictor contest uses 1347 bits of history and 15 tables of mostly 1024 entires per table [91]. Although the global history could be implemented as a circular buffer, which requires only saving and restoring a head pointer, each table requires three circular shift registers of 7 to 16 bits each as recommended by [68]. These snapshot costs are sizable relative to the rest of the predictor. A realistic RTL

[1]This discussion presupposes the ISA is RV64G in which all instructions are 4-bytes in size.

implementation of TAGE is incentivized to either pursue other techniques of checkpointing TAGE state or to pursue many fewer prediction tables.

4.3.5 Implementation Realism

Even if accurate costs of prediction and checkpoint state are taken into account, the design may still fail a realizability test. For example, one branch prediction technique that has been used previously is a *local history* predictor [55]. This requires tracking history on a per branch instruction basis as opposed to the global history tracking branch outcomes on a per program basis. This local history is then used to index into second-level table to make a prediction. However, for a processor fetching four instructions a cycle, this design could require a second-level table with 4 read ports. For this reason, the Alpha EV8 design team discarded the use of a *local history* predictor which had been used in the previous Alpha 21264 design [93].

Branch predictors must also be capable of achieving high clock frequencies and low access latencies, setting further constraints on the organization and algorithmic complexity of the design. The Alpha EV8 team spent considerable effort mapping four separate branch predictors (with a tournament function to choose a winner amongst them) into a design using single-ported, four-way interleaved banks. This required the designers to choose incredibly clever hashing functions that caused each of the four branch predictors to index into the same physical bank and wordline index, while still maintaining enough randomness in the hashed offset to provide the appearance of four separate, uncorrelated branch predictors. Their index hash functions also took into account the criticality of each bit. The bank selector is most critical, so those two bits came straight from the *fetch address* and were unhashed. The wordline index selection was less critical, and was performed using one logic-level of XOR hashing with the global history. Finally, the bit offset index was least critical, and used a larger XOR tree to compute.

4.4 Proposed Improvements for Software Model Evaluations

Although we have strived to make implementing new branch predictor designs in BOOM an approachable project, we recognize the incredible usefulness of C++-based software models at exploring a vast design space, the ease at which new ideas can be implemented, and the simulation speed that is competitive with FPGAs without suffering the FPGA synthesis overhead. To this end, we have a few proposals that may help increase the fidelity of the software-based simulation models without significantly hampering designer productivity.

Not all state bits are equal in cost. Designs should strive to account for the particular implementation technology for their memory arrays. Are they using single-ported or two-ported SRAM? Are these bits stored in flip-flops?

Models can queue up prediction updates and perform them at some later time to mimic the delay in updating the predictor. This delay can be changed to show the sensitivity of a particular design to this update latency.

State checkpointing costs should also be measured. Although traces only provide a path of committed instructions, the simulator could occasionally inject synthetic false paths that then require state restoration when resolving the mispredicted branch. If the designer fails to properly handle checkpoints and restores, the performance of the branch predictor may suffer as it gets polluted by wrong information.

The simulation traces can be augmented with information on which branches may be more critical relative to other branches. For example, a cache model could be used to correlate which load-dependent branches are likely to be more expensive to mispredict due to a cache miss from the load.

Although these proposed changes are not meant to replace the existing "Level 0" unpipelined simulators, they can provide a supplementary "Level 1" model that attempts to provide a more accurate gauge of a branch predictor's effect on performance, power, and area. Valuable future work would be to find exactly where these Level 0 and Level 1 models can lead to inconsistent qualitative results in providing relative rankings of different designs.

4.5 Conclusion

Branch prediction research has been an important and well-studied field for over 30 years. Trace-based, one-at-a-time, unpipelined simulations have allowed researchers to focus on quickly exploring the effectiveness of high-level algorithms to find the upper-bound performance. However, RTL implementers must bring their own creativity to the table in mapping well-performing models to RTL. This chapter discusses some of the gaps between models and RTL and how BOOM's implementation of two different branch predictors address these challenges.

There were a number of challenges in implementing a branch predictor at the RTL level. Superscalar prediction and the disconnect between instructions and *fetch packets*, mixed with the delayed updating of the global history register, meant that multiple executions of the same code path might not see consistent behavior from run to run due to icache misses, TLB misses, backend stalls, and branch mispredictions.

The delayed history register behavior was unbelievably difficult to handle. A "shadow" existed where branches making predictions used a global history that did not reflect the direction of the branch one or two cycles in front of it. When a branch mispredicted, the global history has to be reset and corrected to reflect the true program's global history which must accurately capture this "shadow." Path-based prediction [73] may prove to be a more effective means of tracking program behavior.

Updating the predictor speculatively, maintaining predictor snapshots, and resetting the predictor on mispredictions or pipeline exceptions is a complexity not explored in unpipelined simulators. This complexity changes the cost models of different predictor configurations.

Branch predictors have the nice property that correctness is not required, but the downside is that they are more difficult to test for performance degradation. A set of simple functional tests were used to look for performance regressions, but more complex interactions are difficult to verify.

Despite these challenges, a synthesizable branch predictor has been successfully implemented for BOOM. Using a geometric mean across 9 SPECint benchmarks, BOOM using a 15-bit gshare predictor has a branch misprediction rate of 8.9%, which is half as many mispredictions as the Cortex-A9 with 19.0%. A 13-bit gshare predictor, implemented using three 2kB single-port SRAMs, was taped out as part of the BROOM test chip.

Chapter 5

Describing an Out-of-order Execution Pipeline Generator

The execution pipeline resides in the core's "backend" and is where operations, as described by the instructions that have been fetched by the frontend, are carried out. For a modern out-of-order core, these operations are executed speculatively (the frontend may have fetched the wrong instructions) and in the order of their dependences (and not in the order they were fetched).

The execution pipeline contains many different *functional units*, each supporting a different set of operations such as arithmetic computations, memory system requests, or changes to the control flow of the instruction stream.

The execution datapath can vary widely between processors in both the number and the mix of the provided functional units. A Cortex-A15 can issue up to eight micro-ops per cycle onto eight different functional units — two integer arithmetic/logic units (ALUs), an integer multiplier, two floating-point/NEON vector units, a branch unit, a memory load unit, and a memory store unit. The pipeline depth of each functional unit of the A15 varies from three cycles for an integer ALU unit to twelve cycles for a floating-point unit [113]. Similarly, the Samsung M1 is able to issue up to nine micro-ops per cycle — it adds an additional issue scheduler for store data generation [18]. Table 5.1 shows a comparison of different industry processor datapaths.

The goal of this chapter is to provide insight into how BOOM's execution pipeline is mapped to *Chisel* RTL and how leveraging good software engineering practices has allowed us to describe a parameterizable pipeline generator that can be extended with additional functionality relatively quickly. More importantly, we have been able to utilize expert-written hardware units designed for use in a different context by encapsulating them in a manner that allows them to operate within a parameterized and highly speculative out-of-order pipeline. By implementing a pipeline generator for BOOM's out-of-order execution pipeline and utilizing expert-written hardware, we were able to quickly implement RISC instructions in BOOM as well as pursue more radical design changes such as those discussed in Chapter 6.

This chapter first details the design goals and challenges behind the implementation of BOOM's execution datapath in Section 5.1, followed by a description of the implementation in Section 5.2. Section 5.3 then presents a small case study describing how adding support for single- and double-precision floating point was aided by BOOM's generator design. Section 5.4 walks through an example of adding a new instruction to BOOM's pipeline. Section 5.5 discusses some of the limitations of the generator design and Section 5.6 concludes.

5.1 Goals and Challenges

As a processor generator, BOOM needs the flexibility to provide the designer a range of execution datapath options. If BOOM can generate a variety of datapath designs, the designer can then explore the design space via simulation to find the optimal functional unit mix that provides the best performance for a particular set of workloads. In order to achieve this goal, the execution datapath must be *parameterizable*. However, this flexibility comes with challenges.

The first challenge is the wide reach of the execution pipeline. It interacts with many disparate parts of the processor core and making the execution pipeline flexible requires making all of the other units it interacts with flexible as well. Adding additional functional units typically requires adding more issue ports to the issue windows and more read and write ports to the register file.

Another challenge is that functional units are incredibly complex units that are carefully crafted to provide high performance within a given area constraint. Therefore, it is highly desirable to reuse expert-written functional units that were likely implemented with different micro-architectures in mind. BOOM makes use of the `berkeley-hardfloat` repository [114], which provides a set of expert-written floating-point units designed for the in-order Rocket processor and Rocket-compatible vector units. However, the `hardfloat` units are designed under the assumption that all operations the units receive are committed and will modify the architectural state of the processor. This assumption is at odds with the highly speculative, run-ahead nature of an out-of-order processor like BOOM.

In summary, the two main design challenges are:

1. Providing flexibility to change the number and mix of functional units in the execution datapath.

2. Being able to leverage expert-written functional units that may not perfectly match an out-of-order pipeline.

To address this challenge, BOOM has a notion of an abstract `FunctionalUnit` class that provides a generic interface to the rest of the execution datapath. Expert-written functional units can then be encapsulated within the `FunctionalUnit` wrapper. The wrapper handles the meta-data needed to track the speculation state of each micro-op occupying the functional unit's pipeline and suppresses write-back as needed once each micro-op finishes execution.

Table 5.1: A survey of industry execution datapaths and the different functional units available. There is no consensus on the exact number and mix of functional units provided. More functional units allows for more simultaneous operations that improves performance, but more units require more issue ports and register file ports, which are expensive commodities.

	MIPS R10K [124]	Cortex-A9 [44]	Cortex-A15 [113]	Samsung M1 [18]
Year	1996	2007	2010	2016
Max Issue Width	5	4	8	9
Dedicated Branch Unit	0	0	1	1
Simple Integer Unit	1 ALU+Br	1 ALU+Br	2 ALU	2 ALU
Complex Integer Unit	1 iMul/iDiv	1 iMul	1 iMul	1 ALUC/iMul/iDiv
Floating Point Unit	1+1 FAdd+FMul	1 FP/NEON	2 FP/NEON	1 + 1 FMul/SIMD+FAdd/SIMD
Memory Unit	1 Load/Store	1 Load/Store	1+1 Load+Store	1+1+1 Load+Store AddrGen +Store DataGen

This functional unit abstraction can be further extended; an abstract `ExecutionUnit` can encapsulate many functional units and provide a generic interface to the processor's issue ports, register file ports, and load/store unit ports. The next section, Section 5.2, goes into more detail on the implementation of BOOM's execution datapath and how it addresses the challenges raised here.

5.2 The BOOM Execution Pipeline

The execution pipeline covers the execution and write-back of micro-ops. Although the micro-ops will travel down the pipeline one after the other (in the order they have been issued), the micro-ops themselves are likely to have been issued to the execution pipeline out-of-order. Figure 5.1 shows an example execution pipeline for a dual-issue BOOM.

5.2.1 Branch Speculation

All micro-ops that are "inflight" in BOOM are given a *branch mask*, where each bit in the *branch mask* corresponds to an un-executed, inflight branch that the micro-op is speculated under. Each branch in *Decode* is allocated a *branch tag*, and all following micro-ops will have the corresponding bit in the *branch mask* set (until the branch is resolved by the Branch Resolution Unit).

If the branches (or jumps) have been correctly speculated by the front-end, then the Branch Resolution Unit's only action is to broadcast the corresponding branch tag to *all* inflight micro-ops that the branch has been resolved correctly. Each micro-op can then clear the corresponding bit in its *branch mask*, and that branch tag can then be allocated to a new branch in the *Decode* stage.

If a branch (or jump) is misspeculated, the Branch Resolution Unit must redirect the PC to the correct target, clear the front-end and fetch buffer, and broadcast the misspeculated *branch tag* so that all dependent, inflight micro-ops may be killed.

These branch resolution signals are broadcast to all functional units which must track the speculation status of each micro-op under its stewardship and suppress the write-back of any killed micro-ops.

5.2.2 Execution Units

An Execution Unit is a module that a single issue port will schedule micro-ops onto and contains some mix of functional units. Phrased in another way, each issue port from the issue window talks to one and only one Execution Unit. An Execution Unit may contain just a single simple integer ALU, or it could contain a full complement of floating point units, a integer ALU, and an integer multiply unit.

An Execution Unit provides a bit-vector of the functional units it has available to the issue scheduler. The issue scheduler will only schedule micro-ops that the Execution Unit

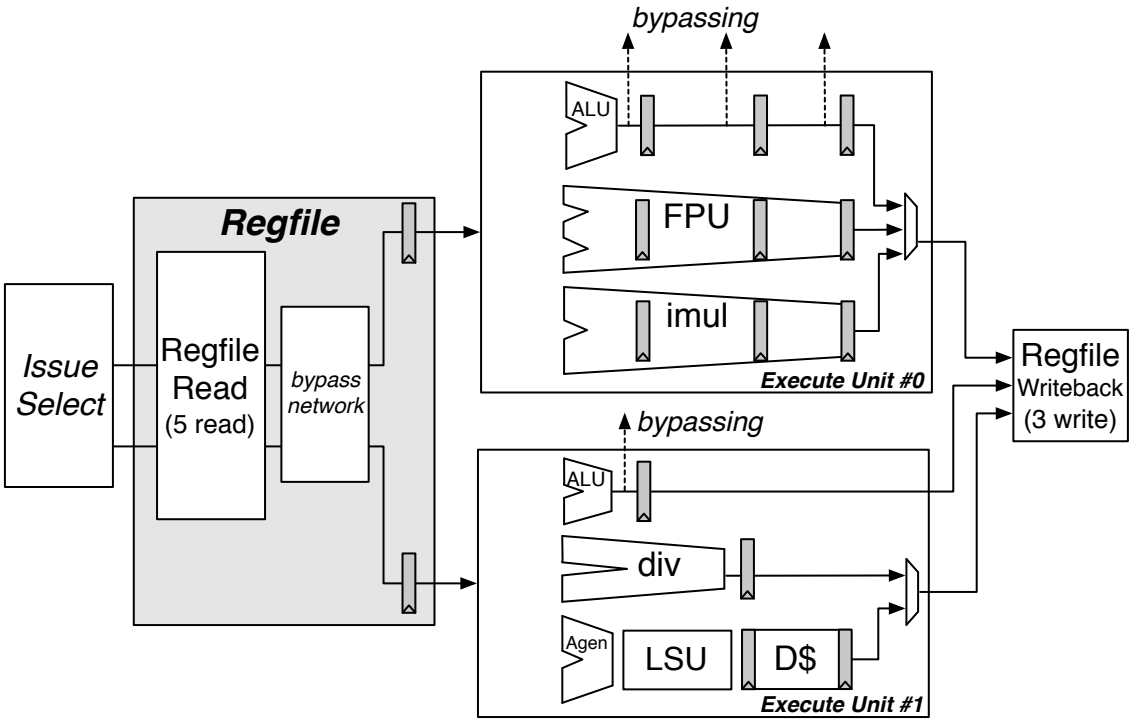


Figure 5.1: An example pipeline for a dual-issue BOOM. The first issue port schedules micro-ops onto Execute Unit #0, which can accept ALU operations, FPU operations, and integer multiply operations. The second issue port schedules ALU operations, integer divide instructions (unpipelined), and load/store operations. The ALU operations can bypass to dependent instructions. Note that the ALU in EU#0 is padded with pipeline registers to match latencies with the FPU and iMul units to make scheduling for the write-port trivial. Each Execution Unit has a single issue-port dedicated to it but contains within it a number of lower-level Functional Units.

```
1  val exe_units = ArrayBuffer[ExecutionUnit]()
2
3  exe_units += Module(new ALUExeUnit(is_branch_unit   = true,
4                                     has_fpu          = true,
5                                     has_mul          = true,
6                                     shares_csr_wport = true
7                                    ))
8  exe_units += Module(new ALUExeUnit(has_div = true
9                                    ))
10 exe_units += Module(new MemExeUnit())
```

Code 5.1: Describing the pipeline in *Chisel* code. Additional Execution Units may be added to `exe_units` and the connections to the register file and issue window will be handled automatically.

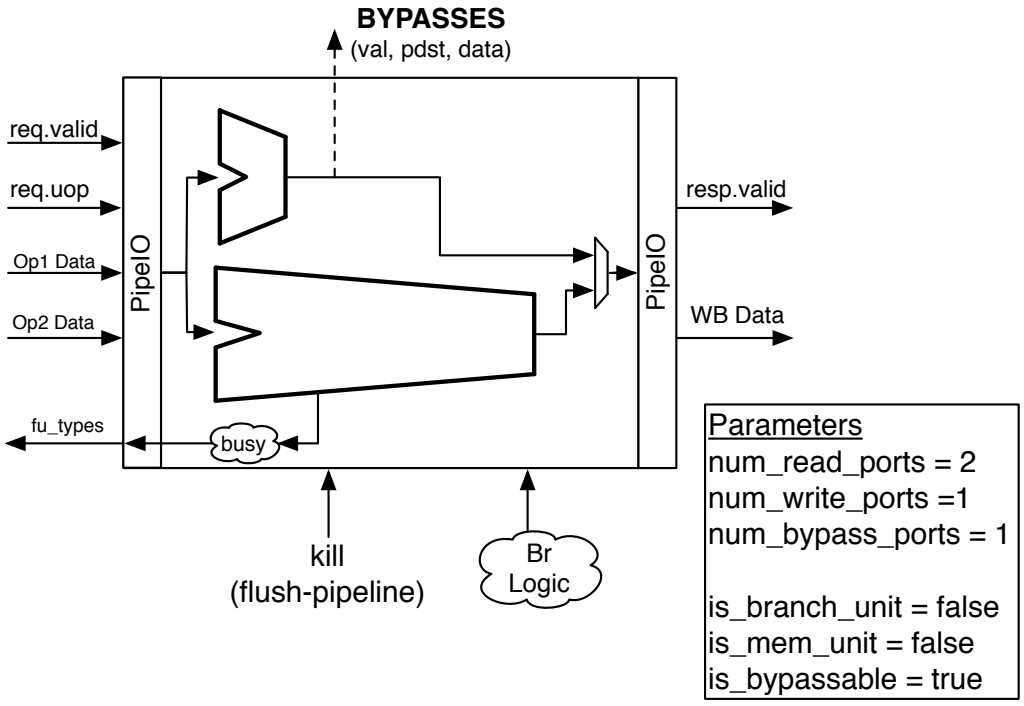


Figure 5.2: An example Execution Unit. This particular example shows an integer ALU (that can bypass results to dependent instructions) and an unpipelined divider that becomes *busy* during operation. Both functional units share a single write-port. The Execution Unit accepts both *kill* signals and *branch resolution* signals and passes them to the internal functional units as required.

supports. For functional units that may not always be ready (e.g., an iterative divider), the appropriate bit in the bit-vector will be disabled (see Fig 5.2).

The abstract Execution Unit module provides a flexible abstraction which gives a lot of control over what kind of Execution Units the architect can add to their pipeline. The rest of the processor pipeline – the issue ports, the register file ports, and the load/store unit ports – only need to interface directly with the abstract Execution Unit input/output interface.

Code 5.1 shows the top-level instantiation of the execution pipeline. An `ArrayBuffer` of `ExecutionUnit`s are constructed. The register file ports and issue select ports are auto-generated to interface with the `exe_units` array. Constructor arguments to each Execution Unit enables the desired functional units within each Execution Unit.

5.2.3 Functional Units

Functional units are the muscle of the CPU, computing the necessary operations as required by the instructions. Functional units typically require a knowledgable domain expert to implement them correctly and efficiently.

For this reason, BOOM uses the expert-written, low-level functional units from the Rocket and Berkeley `hardfloat` repositories [83, 114]. However, the expert-written functional units created for the Rocket in-order processor make assumptions about in-order issue and commit points – namely, that once an instruction has been dispatched to them it will never need to be killed. These assumptions break down for out-of-order processors that derive a lot of their performance by speculatively running ahead and executing instructions that may need to be killed.

Instead of attempting to re-write the functional units to understand the semantics of a speculative and out-of-order pipeline, BOOM provides an abstract `FunctionalUnit` class (see Fig 5.3) that "wraps" the lower-level functional units with the parameterized auto-generated support code needed to make them work within BOOM. The request and response ports are abstracted, allowing functional units to provide a unified, interchangeable interface. Each functional unit also provides a set of configurable parameters allowing the wrapper to match the underlying characteristics of the low-level functional unit.

Figure 5.5 shows the full hierarchy of functional units and their abstract helper modules. From the abstract `FunctionalUnit` class, functional units are further broken down into `Pipelined` and `Iterative` units.

5.2.3.1 Pipelined Functional Units

A pipelined functional unit can accept a new micro-op every cycle, and each micro-op will take a known, fixed latency.

Speculation support is provided by auto-generating a pipeline that passes down the micro-op meta-data and *branch mask* in parallel with the micro-op within the expert-written functional unit. If a micro-op is misspeculated, it's response is de-asserted as it exits the functional unit. Fig 5.3 shows an example pipelined functional unit.

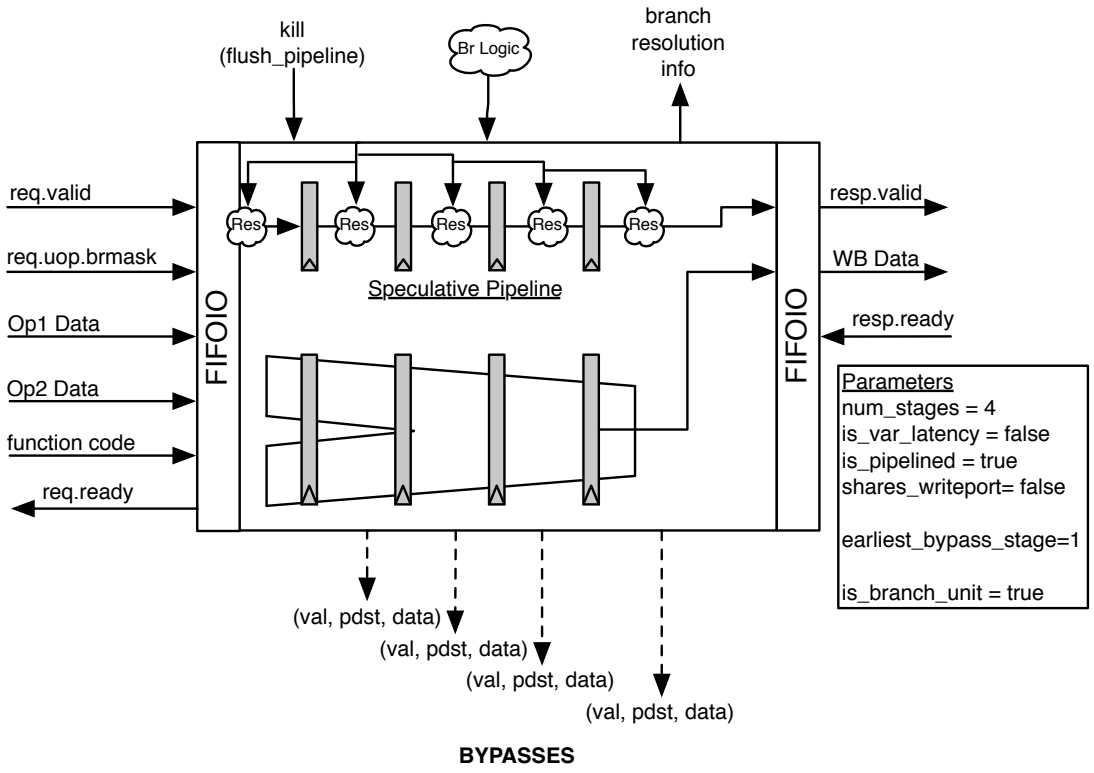


Figure 5.3: The abstract Pipelined Functional Unit class. An expert-written, low-level functional unit is instantiated within the Functional Unit. The request and response ports are abstracted and bypass and branch speculation support is provided. Micro-ops are individually killed by gating off their response as they exit the low-level functional unit.

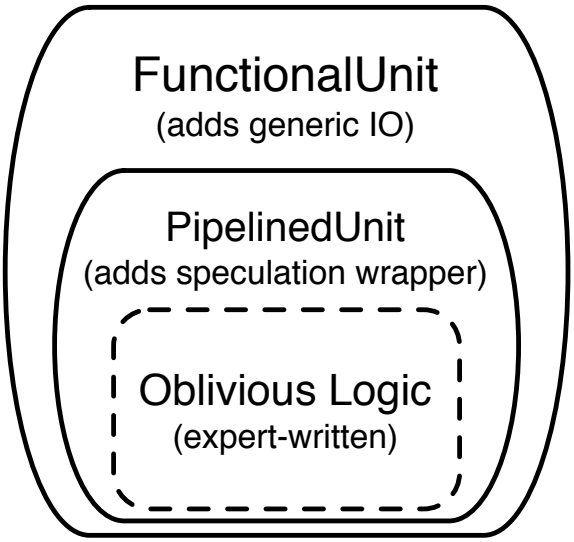


Figure 5.4: The functional unit abstraction allows for the easy encapsulation of expert-written functional unit logic. The expert-written logic is instantiated by a `PipelinedUnit`, which adds auto-generated speculation support logic. The encompassing `FunctionalUnit` abstract class provides a generic I/O interface into the unit that allows the rest of the BOOM pipeline to interface with any functional unit.

5.2.3.2 Iterative Functional Units

Iterative functional units take a variable (and unknown in advance) number of cycles to complete a single operation. Once occupied, they de-assert their ready signal and no additional micro-ops may be scheduled to them. An example iterative unit is an integer divider.

Speculation support is provided by tracking the *branch mask* of the micro-op in the functional unit. An expert-written iterative functional unit should provide a *kill* signal to quickly remove misspeculated micro-ops. BOOM currently only supports single-occupancy in its iterative functional units.

5.2.4 The Load/Store Unit

The Load/Store Unit (LSU) is a challenging unit to encapsulate. A pipelined `MemAddrCalc` unit is exposed as an Execution Unit to the issue ports for scheduling load and store micro-ops. However, the LSU differs from other pipelined functional units in a number of ways. First, it needs a connection to the L1 data cache which can invoke architecturally-visible changes. Second, load instructions exhibit a variable execution latency due to cache misses and structural hazards in the data cache.

5.3 Case Study: Adding Floating-point Support

This section highlights the advantages to the implementation strategy behind BOOM's execution datapath by discussing the success in adding floating-point instruction support to BOOM over a period of twelve days and in 1092 lines of code.

The RISC-V ISA supports single (F) and double-precision (D) floating-point operations that handle operations that act on a new set of 32 floating-point registers. The F and D extensions also add a new floating-point control and status register (`fcsr`), which stores the current dynamic rounding mode, and the accrued exception state (`fflags`). Data can be moved into and out of the floating-point register file via load and store instructions or by moving values to and from the integer register file.

The RV64FD ISA supports the IEEE 754-2008 floating-point standard and includes support for fused multiply-adds, divisions, and square roots. IEEE 754 floating-point comes with a number of challenges not faced with integer-only pipelines including:

1. support for a separate floating-point register file
2. new special-case values such as infinities, NaNs (not-a-number), and subnormals
3. accrued exceptions to track the occurrence of overflow, underflow, division by zero, invalid operation, and inexact result
4. multiple rounding modes such as round-to-nearest-ties-to-even (RNE) and round-towards-zero (RTZ).

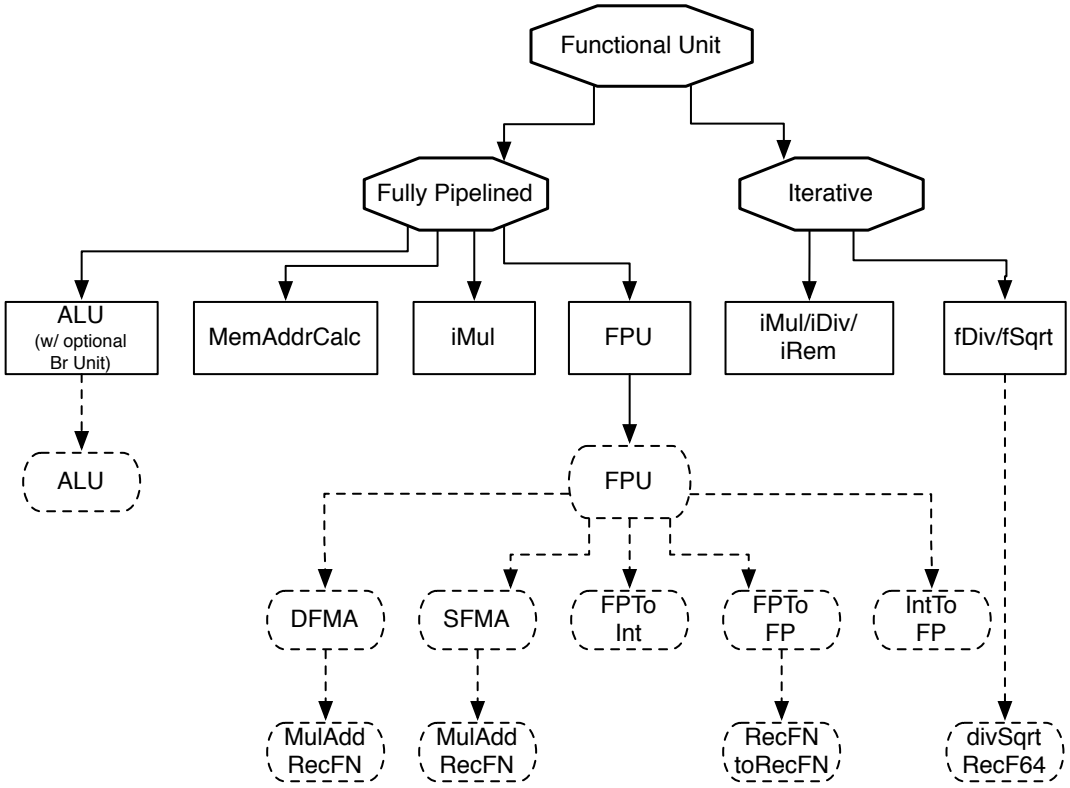


Figure 5.5: The Functional Unit class hierarchy. The dashed ovals are the low-level functional units written by experts, the squares are concrete classes that instantiate the low-level functional units, and the octagons are abstract classes that provide generic speculation support and interfacing with the BOOM pipeline.

The `berkeley-hardfloat` repository [114] provides a library of parameterizable hardware floating-point units written in *Chisel*. The included units provide support for fused multiply-add, divide, square root, conversion between integer and floating-point values, and conversion between floating-point values of differing precision. The implementations of the hardware units total roughly 3,800 lines of code. The widths of the exponent and significand are parameterizable. The floating-point units intending to be pipelined are implemented as combinational logic; the instantiating module must add registers behind the floating-point logic and use the synthesis tool's register retiming logic to effect the desired pipeline latency.

5.3.1 Register File and Register Renaming

In adding floating-point support, we decided to use the existing physical register file to hold both integer and floating-point values. We augmented the register renaming stage to treat the 32 floating-point registers as an extension of the existing 32 integer registers by mapping f0 through f31 to logical registers 32 through 63. This technique allows the busy table, rename map tables, free list register allocator, and register wakeup logic to come "for free". None of these structures have any knowledge of integer versus floating-point but instead see the integer registers x0 through x31 as logical registers 0 through 31 and the floating-point registers f0 through f31 as logical registers 32 through 63. All physical registers were expanded to 65-bits as the `berkeley-hardfloat` units are designed to handle an internal 65-bit format in which the exponent has an additional bit to handle subnormal numbers more efficiently.

Chapter 6 discusses later efforts to separate the floating-point registers from the integer registers to aid in physical design.

5.3.2 Issue Window

We also continued using a single unified issue window to store all instructions. The pipelined floating-point unit (FPU), which handles fused multiply-add operations, conversions, and moves, was placed into the same execution units with other integer functional units. As such, FMA operations used the same issue select port and register file ports as the pipelined integer ALU and integer multiplier units (see Figure 5.1). To make the scheduling of the write ports easier, all functional units within an execute unit are padded to the same latency. For example, the integer ALU now matches the latency of the 3-cycle double-precision FMA unit. To prevent any performance degradation, the ALU is able to bypass its value to any dependent operations until it is written back to the register file.

5.3.3 Floating-point Control and Status Register (`fcsr`)

The RISC-V F and D extensions add accrued exception state. Many floating-point operations may set an exception condition code upon completion. However, no exception is taken. Instead, the programmer must query the floating-point status register to view the current

state of the exception condition codes. Each FPU has a port into the re-order buffer (ROB) to track its exception state. Upon commit, the exception state is accrued into the floating-point status register. When a CSR read instruction for the floating-point status register is encountered, the pipeline is serialized.

Both static and dynamic rounding modes are supported. Static rounding modes are embedded in the instruction and are passed down the pipeline as part of the micro-op meta-data. The dynamic rounding mode is set as part of the floating-point control/status register. Changes to `fcsr` serializes the pipeline.

5.3.4 Hardfloat and Low-level Instantiations

The `berkeley-hardfloat` repository provides a library of low-level building blocks encompassing 3,800 lines of *Chisel* code. The *Rocket-chip* processor library instantiates these blocks and provides the control logic and pipeline registers to build a fully functioning floating-point pipeline. Code 5.2 shows the instantiation of the FMA pipeline. Notice that this FMA pipeline has no knowledge of speculation or out-of-order execution. Any requests sent to the FMA pipeline will return a fixed `latency` cycles later and cannot be killed.

BOOM then bundles the FMA pipeline and other floating-point pipelines to implement a full floating-point unit (FPU). The bundled units include a single-precision FMA pipeline, a double-precision FMA pipeline, an Int-to-FP conversion unit, an FP-to-Int conversion unit, and an FP-to-FP conversion unit. This FPU unit, including all of the control logic, is 203 lines of code. To make scheduling the write port easier, all operations use the same latency. This unit also has no knowledge of speculation.

Table 5.2: The hierarchy from an abstract `FunctionalUnit` to an expert-written fused multply-add block. The lower level units originate from other repositories.

Repository	Unit	lines of code
BOOM	FunctionalUnit (abstract)	20
BOOM	PipelinedFunctionalUnit (abstract)	69
BOOM	FPUUnit (speculation wrapper)	10
BOOM	FPU (no speculation support)	203
Rocket	FP pipelines (e.g., DFMA Pipeline)	274
hardfloat	hardfloat blocks (e.g., mulAddSubRecodedFloatN)	2,061

5.3.5 Pipelined Functional Unit Wrapper

As the FMA pipeline and the FPU that instantiates it has no knowledge of speculation, we must encapsulate them in a `PipelinedFuntionalUnit` to provide the necessary speculation

```
1  class FPUFMAPipe(val latency: Int, sigWidth: Int, expWidth: Int) extends Module
2  {
3    val io = new Bundle {
4      val in = Valid(new FPInput).flip
5      val out = Valid(new FPResult)
6    }
7
8    val width = sigWidth + expWidth
9    val one = UInt(1) << (width-1)
10   val zero = (io.in.bits.in1(width) ^ io.in.bits.in2(width)) << width
11
12   val valid = Reg(next=io.in.valid)
13   val in = Reg(new FPInput)
14   when (io.in.valid) {
15     in := io.in.bits
16     val cmd_fma = io.in.bits.ren3
17     val cmd_addsub = io.in.bits.swap23
18     in.cmd := Cat(io.in.bits.cmd(1) & (cmd_fma || cmd_addsub), io.in.bits.cmd(0))
19     when (cmd_addsub) { in.in2 := one }
20     unless (cmd_fma || cmd_addsub) { in.in3 := zero }
21   }
22
23   val fma = Module(new hardfloat.mulAddSubRecodedFloatN(sigWidth, expWidth))
24   fma.io.op := in.cmd
25   fma.io.roundingMode := in.rm
26   fma.io.a := in.in1
27   fma.io.b := in.in2
28   fma.io.c := in.in3
29
30   val res = new FPResult
31   res.data := fma.io.out
32   res.exc := fma.io.exceptionFlags
33   io.out := Pipe(valid, res, latency-1) // <<--- register re-timing
34 }
```

Code 5.2: The code for the FMA pipeline is shown that instantiates a `berkeley-hardfloat` `mulAddSubRecodedFloatN` module for describing a fused multiply-add unit. Notice line 33 pads the latency of the unit and uses register retiming to implement a parameterized latency, pipelined FMA unit.

```
1  class FPUUnit(num_stages: Int) extends PipelinedFunctionalUnit(
2     num_stages = num_stages,
3     num_bypass_stages = 0,
4     earliest_bypass_stage = 0,
5     data_width = 65)
6     with BOOMCoreParameters
7  {
8     val fpu = Module(new FPU())
9     fpu.io.req <> io.req
10    fpu.io.req.bits.fcsr_rm := io.fcsr_rm
11    io.resp <> fpu.io.resp
12    io.resp.bits.fflags.bits.uop := io.resp.bits.uop
13 }
```

Code 5.3: Wrapping the FPU in the pipelined functional unit wrapper.

support. This encapsulation is straightforward, as the abstract class is doing all of the heavy work of auto-generating the supporting speculation hardware and tracking of the inflight micro-ops and branch masks. As shown in Code 5.3, this takes 10 lines of code. Now, micro-ops may be executed on this unit out-of-order and any micro-ops killed by misspeculation will have their writes suppressed while leaving the other micro-ops in the pipeline untouched. Table 5.2 shows the entire hierarchy from `berkeley-hardfloat` to BOOM's abstract class `FunctionalUnit`, complete with a breakdown of the lines of code for each level in the hierarchy.

5.3.6 Adding the FPU to an Execution Unit

The final step of adding an expert-written unit to BOOM is to take the `PipelinedFunctionalUnit` and instantiate it in an Execution Unit. The Execution Unit is a collection of functional units that share a single issue select port and a set of register file access read and write ports. To make adding floating-point support to BOOM easy, we added the FPUs to existing integer Execution Units. This approach was made possible by placing the floating-point instructions into the same issue window and using the same physical register file as the integer instructions. Code 5.4 shows the necessary code added to an existing Execution Unit to instantiate an FPU and have it share the existing request port and response port with the other functional units.

5.3.7 Results

Figure 5.6 shows how quickly floating-point support was added based on the git commit history. At the end of two weeks, we could describe somewhat arbitrary superscalar, out-of-order datapaths with floating-point units. We could parameterize the latency of the FPUs and the number of FPUs. In changing these configurations, the issue ports, register file ports, rename logic, wakeup logic, and more are auto-generated to interface with the new

```
1    // The list of all functional units in this ExecutionUnit
2    val fu_units = ArrayBuffer[FunctionalUnit]()
3    ...
4
5    // FPU Unit -----------------------
6    var fpu: FPUUnit = null
7    if (has_fpu)
8    {
9       fpu = Module(new FPUUnit())
10      fpu.io.req.valid           := io.req.valid && io.req.bits.uop.fu_code_is(FU_FPU)
11      fpu.io.req.bits.uop        := io.req.bits.uop
12      fpu.io.req.bits.rs1_data   := io.req.bits.rs1_data
13      fpu.io.req.bits.rs2_data   := io.req.bits.rs2_data
14      fpu.io.req.bits.rs3_data   := io.req.bits.rs3_data
15      fpu.io.req.bits.kill       := io.req.bits.kill
16      fpu.io.fcsr_rm             := io.fcsr_rm
17      fpu.io.brinfo <> io.brinfo
18
19      fu_units += fpu
20   }
21   ...
22
23   io.resp(0).valid :=
24      fu_units.map(_.io.resp.valid).reduce(_|_)
25   io.resp(0).bits.uop :=
26      new MicroOp().fromBits(
27         PriorityMux(fu_units.map(f => (f.io.resp.valid, f.io.resp.bits.uop.toBits))))
28   io.resp(0).bits.data :=
29      PriorityMux(fu_units.map(f => (f.io.resp.valid, f.io.resp.bits.data.toBits))).toBits
```

Code 5.4: Instantiating the FPU within an Execution Unit.

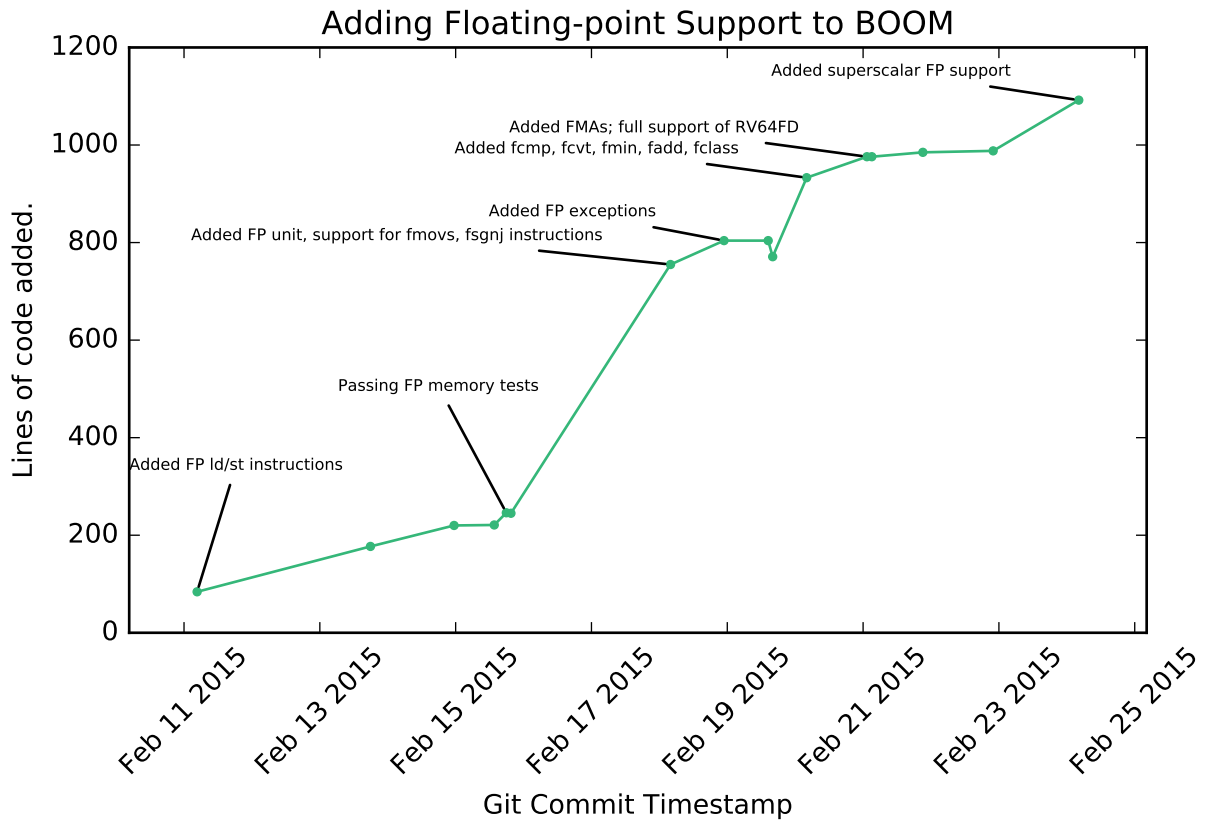


Figure 5.6: Support for the RISC-V single-("F") and double-("D") precision extensions was implemented over a two week period.

configuration. This effort was made easier by having a good abstraction framework developed and having an open-source repository of expert-written functional units to utilize.

THis two week case study of adding floating-point support was only an initial effort. Chapter 6 will discuss a later effort to improve synthesis quality-of-result by splitting the integer and floating-point registers into separate register files and the splitting of the issue window into three separate issue windows for integer, memory, and floating-point operations.

5.4 Case Study: Adding a Binary Manipulation Instruction

Provided the framework discussed in the previous sections, we can quickly add new instructions to BOOM that do not deviate too far from standard RISC-like micro-ops.

A proposed extension to the RISC-V ISA is the Binary manipulation extension "B". This extension may include rotates, byte swaps, counting leading zeroes, and population count. The `popcnt` instruction returns the number of 1s in a register. This instruction was added to the x86 ISA as part of the `SSE2` extension and was measured to take 3 cycles on an Intel Skylake core with a throughput of 1 instruction per cycle [36].

A potential implementation of `popcnt` can be described using fully combinational logic and then padded with two registers and retimed to provide three cycles to complete the computation while also allowing a pipeline throughput of one operation per cycle.

5.4.1 Decode, Rename, and Instruction Steering

First, our instruction must be added to the decode table in the *Decode* stage (Code 5.5). In particular, we must specify the type of registers used by the instruction and which functional unit the instruction should be steered towards. The register types help guide register renaming and ensures we write back to the correct register file. Instructions that want to add additional operations to existing functional units can continue to use an existing functional unit's code. For example, saturating arithmetic will want to use the existing *arithmetic logic unit (ALU)* path.

5.4.2 The Popcount Unit Implementation

We then need to implement the popcount functional unit. This unit will perform a popcount operation with the specified latency and throughput but otherwise have no support for speculation. Code 5.6 shows an implementation of a `popcount` unit.

Next, we encapsulate the low-level popcount unit with the `PipelinedFunctionalUnit` abstract class to provide the speculation support necessary for speculative, out-of-order execution as shown in Code 5.7.

Next, we need to instantiate this `PipelinedPopcntUnit` functional unit within an Execution Unit as shown in Code 5.8. We can place the `popcount` functional unit within the

```
1          //                                                                  frs3_en
2          //     is val inst?                                                 |  imm sel
3          //     |  is fp inst?                                              |  |
4          //     |  |  is single-prec?                    rs1 regtype       |  |
5          //     |  |  |  micro-code                      |       rs2 type|  |
6          //     |  |  |  |         iq-type  func unit     |       |       |  |
7          //     |  |  |  |         |        |               |       |       |  |
8          //     |  |  |  |         |        |        dst    |       |       |  |
9          //     |  |  |  |         |        |        regtype|       |       |  |
10         //     |  |  |  |         |        |        |      |       |       |  |
11 val table: Array[(BitPat, List[BitPat])] = Array(//  |      |       |       |  |
12
13 POPC    -> List(Y, N, X, uopPOPC , IQT_INT, FU_POPC, RT_FIX, RT_FIX, RT_X  , N, IS_X,
14
15         //                            bypassable (aka, known/fixed latency)
16         //     is_load                       |  br/jmp
17         //     |  is_store                   |  |  is jal
18         //     |  |  is_amo                  |  |  |  allocate_brtag
19         //     |  |  |  is_fence             |  |  |  |
20         //     |  |  |  |  is_fencei         |  |  |  |
21         //     |  |  |  |  |  mem    mem     |  |  |  |  is unique? (clear pipeline first)
22         //     |  |  |  |  |  cmd    msk     |  |  |  |  |  flush on commit
23         //     |  |  |  |  |  |      |       |  |  |  |  |  |  csr cmd
24         //     |  |  |  |  |  |      |       |  |  |  |  |  |  |
25
26               N, N, N, N, N, M_X  , MSK_X , N, N, N, N, N, N, CSR.N),
27   ...
```

Code 5.5: The decode table entry for `popcount`. The most relevant part is marking the register types and the functional unit type (`FU_POPC`).

```
1  class PopCntUnit(num_stages: Int) extends Module
2  {
3     val io = new Bundle {
4        val valid = Bool(INPUT)
5        val dw  = Bool(INPUT)
6        val in0 = UInt(INPUT, 64)
7        val out = UInt(OUTPUT, 64)
8     }
9
10    val result =
11       Mux(io.dw,
12          PopCount(io.in0),
13          PopCount(io.in0(31,0)))
14
15    io.out := Pipe(io.valid, result, num_stages).bits
16 }
```

Code 5.6: The "expert-written" popcount functional unit. It has no knowledge of speculation or out-of-order execution and provides no ability to kill any particular micro-op currently occupying the popcount unit. The `PopCount` construct in *Chisel* combinationally sums the bits in a signal. The latency is padded to `num_stages` by using the `Pipe` construct which adds registers to effect the requested latency. Synthesis tools can later use *register retiming* to balance the popcount logic as needed.

```
1
2  class PipelinedPopcntUnit(implicit p: Parametrs) extends PipelinedFunctionalUnit(
3     num_stages, p(tile.TileKey).core.popc.latency,
4     num_bypass_stages = 0,
5     earliest_bypass_stage = 0,
6     data_width = 64)(p)
7  {
8     val pcu = Module(new PopCntUnit)
9     pcu.io.valid := io.req.valid
10    pcu.io.in0 := io.req.bits.rs1_data
11    pcu.io.in1 := io.req.bits.rs2_data
12    pcu.io.dw := io.req.bits.uop.ctrl.fcn_dw
13
14    io.resp.bits.data := pcu.io.out
15 }
```

Code 5.7: The "expert-written" popcount functional unit is then encapsulated by the `PipelinedFunctionalUnit` abstract class that provides the speculation support and interfacing with the rest of the execution pipeline.

```
class ALUExeUnit(
   ...
   use_popc : Boolean = false)
   (implicit p: Parameters)
   extends ExecutionUnit(
      ...
      has_popc = has_popc)(p)
{
   ...

   // signal to the issue select port via the fu_types bit vector
   // that we support popcnt operations.
   io.fu_types := FU_ALU |
      ...
      Mux(Bool(has_popc), FU_POPC, 0.U)

   // The list of all functional units in this ExecutionUnit
   val fu_units = ArrayBuffer[FunctionalUnit]()
   ...

   // POPCNT Unit ------------------------
   var pcu: PipelinedPopcntUnit = null
   if (has_popc)
   {
      pcu = Module(new PipelinedPopcntUnit())
      pcu.io.req.valid           := io.req.valid && io.req.bits.uop.fu_code_is(FU_POPC)
      pcu.io.req.bits.uop        := io.req.bits.uop
      pcu.io.req.bits.rs1_data   := io.req.bits.rs1_data
      pcu.io.req.bits.rs2_data   := io.req.bits.rs2_data
      pcu.io.req.bits.kill       := io.req.bits.kill
      pcu.io.brinfo <> io.brinfo

      fu_units += pcu
   }
   ...
```

Code 5.8: The popcount unit with speculation support is then instantiated by an Execution Unit, which provides the interfacing to the rest of the execution pipeline. In this example, we have added the `popcount` unit to the existing `ALUExeUnit` and then signaled our support of `popcount` instructions via the `fu_types` bit vector, which provides the issue select port information on what operations this Execution Unit supports.

existing `ALUExeUnit` and provide a new constructor parameter to enable the `popcount` unit as desired. In Code 5.9 we show instantiating the `popcount` unit in an Execution Unit alongside an integer multiplier and an integer ALU.

5.5 Limitations

The organization of BOOM's execution pipeline — focused on supporting RISC instructions – has allowed us to quickly and easily add support for the entire RV64G ISA by reusing expert-written functional units. However, some assumptions have been made and not all

```
1  val exe_units = ArrayBuffer[ExecutionUnit]()
2
3  exe_units += Module(new ALUExeUnit(is_branch_unit   = true,
4                                     has_fpu          = true,
5                                     shares_csr_wport = true
6                                    ))
7  exe_units += Module(new ALUExeUnit(has_div = true
8                                     has_mul = true,
9                                     has_popc = true
10                                   ))
11 exe_units += Module(new MemExeUnit())
```

Code 5.9: Describing the pipeline with a `popcount` unit added to the second `ALUExeUnit`.

types of instruction extensions map well to the current organization.

First, BOOM only supports resolving one branch instruction per cycle. Although this is not an insurmountable restriction, we did not find it necessary to handle a higher throughput of branches.

The data cache that BOOM interfaces with only provides a single 64 bit request port; therefore BOOM only supports instantiating one memory execution unit that can handle one load address or store address computation per cycle. Many high-performance cores are capable of handling two loads per cycle, but that capability comes at a greater complexity and power cost. The modifications to BOOM's execution pipeline to support two loads per cycle would be relatively modest — the main difficulty would be implementing a new data cache that could support two loads simultaneously.

The execution pipeline autogenerates the register file read port connections. However, the read ports on the register file are statically scheduled (each Execution Unit has its own private read ports). For wider processors, this becomes an over-provisioning of a very expensive resource. Many instructions only use one operand. And for instructions that do need operands, [111] notes that 60% of operands are available through the bypass network. However, as BOOM statically schedules the read ports, this sets an upper limit on BOOM's issue width as the number of required register read ports will grow more quickly than if dynamic scheduling techniques were used.

Finally, there are some limitations that make BOOM's execution pipeline organization ill-suited for more CISC-like instructions, in particular, instructions that require interactions with the memory system or introduce stateful side-effects that make out-of-order and speculative run-ahead more difficult. For example, many proposed RISC-V extensions blend the boundaries between processor instructions and *domain-specific accelerators (DSA)*. One challenging example is a *crypto engine* — a DSA that allows the programmer to implement a number of different cipher algorithms [63]. The crypto engine may access memory, which adds both the complication of interfacing with the cache hierarchy and the challenges of throwing page fault exceptions. Crypto engines are also stateful. The engine must be configured, instructions may induce state transitions, and the operating system may need to interact with it differently based on the current privilege mode. Some instructions will write to its own internal registers and other instructions may write back to the integer register file. The engine may also handle some instructions with a fixed latency while other operations may have a variable latency. Finally, crypto engines may have their own resources that need to be allocated. All of these properties make it a challenge to integrate into BOOM's RISC execution pipeline.

5.6 Conclusion

This chapter described the organization of BOOM's superscalar execution pipeline and how its organization was motivated by a desire to build a parameterizable generator and a need to utilize expert-written functional units. This organization has worked well for implementing

instructions that map well to RISC pipelines. More complex instruction extensions that require their own memory port accesses or provide more stateful execution will require a more careful evaluation of how best to implement them within the BOOM processor.

The next chapter, *Chapter 6: VLSI Implementation Effort*, will discuss the changes made to BOOM based on a design exploration performed through synthesis, place, and route using a foundry-provided standard-cell library and memory compiler. The speed that these changes were made, which included splitting apart the register files and adding new functional units to manage moving data between register files, was greatly aided by the design methodology discussed in this chapter.

Chapter 6

VLSI Implementation Effort

On Aug 15, 2017, we taped out an SRAM resiliency test-chip using a BOOM core as the central processing component. This chip was called *BROOM*, for the Berkeley **Resilient** Out of Order Machine. The focus of the BROOM chip was on resiliency techniques that would allow SRAM to both prevent and tolerate higher bit-cell failure rates while running at lower voltages to save power. A number of techniques were explored including *line recycling*, *dynamic column redundancy*, and *tag protection via flip-flop-based bit bypassing*.

While Section 3.5 discusses some of the methodology we followed to tapeout an out-of-order core written in *Chisel* using standard cells, this chapter will focus on the design changes made to BOOM in light of new synthesis, place and route data provided by a foundry-provided standard-cell library and memory compiler. For this tapeout and the associated analysis, we used the TSMC 28 nm HPM process (high performance mobile).

We labeled the culminating design "BOOMv2", an update in which the design effort has been informed by analysis of BOOM in a contemporary industrial tool flow. In contrast, the only synthesis data available during the design evaluation of BOOMv1 was provided through educational libraries that lacked a memory compiler and provided unrealistic timings. We also had access to standard single- and dual-ported memory compilers provided by the foundry, allowing us to explore design trade-offs using different SRAM memories and comparing against synthesized flip-flop arrays. The main distinguishing features of BOOMv2 over BOOMv1 include an updated 3-stage front-end design with a bigger, set-associative Branch Target Buffer (BTB); a pipelined *register rename* stage; separate floating-point and integer register files; a dedicated floating-point pipeline; separate issue windows for floating-point, integer, and memory micro-operations; and separate stages for *issue-select* and *register read*.

Managing the complexity of the register file was the largest obstacle to improving BOOM's clock frequency and manufacturability. We spent considerable effort on place and routing a semi-custom 9-port register file to explore the potential improvements over a fully synthesized design in conjunction with microarchitectural techniques to reduce the size and port count of the register file.

BOOMv2 has a 37 fanout-of-four (FO4) inverter delay after synthesis and 50 FO4 after

place-and-route, a 24% reduction from BOOMv1's 65 FO4 after place-and-route. Unfortunately, instruction per cycle (IPC) performance drops up to 20%, mostly due to the extra latency between load instructions and dependent instructions. However, the new BOOMv2 physical design paves the way for IPC recovery later.

6.1 Background

BOOM was inspired initially by the MIPS R10K and Alpha 21264 processors from the 1990s, whose designs teams provided relatively detailed insight into their processors' microarchitectures [124, 71, 55]. However, both processors relied on custom, dynamic logic which allowed them to achieve very high clock frequencies despite their very short pipelines[1] — the Alpha 21264 has 15 fanout-of-four (FO4)[2] inverter delays [22]. As a comparison, the synthesizable[3] Tensilica's Xtensa processor, fabricated in a 0.25 micron ASIC process and contemporary with the Alpha 21264, was estimated to have roughly 44 FO4 delays [22].

As BOOM is a synthesizable processor, we must rely on microarchitecture-level techniques to address critical paths and add more pipeline stages to trade off instructions per cycle (IPC), cycle time (frequency), and design complexity. We also lack the manpower to carry out many of the custom design techniques to reach the low FO4 of past industry designs. However, as process nodes have become smaller, transistor variability has increased and power-efficiency has become restricting, many of the more aggressive custom techniques have become more difficult and expensive to apply [3]. Modern high-performance processors have largely limited their custom design efforts to more regular structures such as memories and register files.

6.2 BOOMv1

BOOMv1 follows the 6-stage pipeline structure of the MIPS R10K — *fetch*, *decode/rename*, *issue/register-read*, *execute*, *memory*, and *writeback*. During *decode*, instructions are mapped to *micro-operations* (uops) and during *rename* all logical register specifiers are mapped to physical register specifiers. For design simplicity, all uops are placed into a single unified issue window. Likewise, all physical registers (both integer and floating-point registers) are located in a single unified physical register file. Execution Units can contain a mix of integer units and floating-point units. This greatly simplifies floating-point memory instructions and floating-point–integer conversion instructions as they can read their mix of integer and floating-point operands from the same physical register file. BOOMv1 also utilized a short

[1] The R10K has five stages from instruction fetch to integer instruction write-back and the Alpha 21264 has seven stages for the same path. Load instructions take an additional cycle for both processors.

[2] An inverter driving four times its input capacitance. FO4 is a useful, relatively technology-agnostic measurement of a circuit path length.

[3] A "synthesizable" design is one whose gate net list, routing, and placement is generated nearly exclusively by CAD tools. For comparison, "custom" design is human-created logic design and placement.

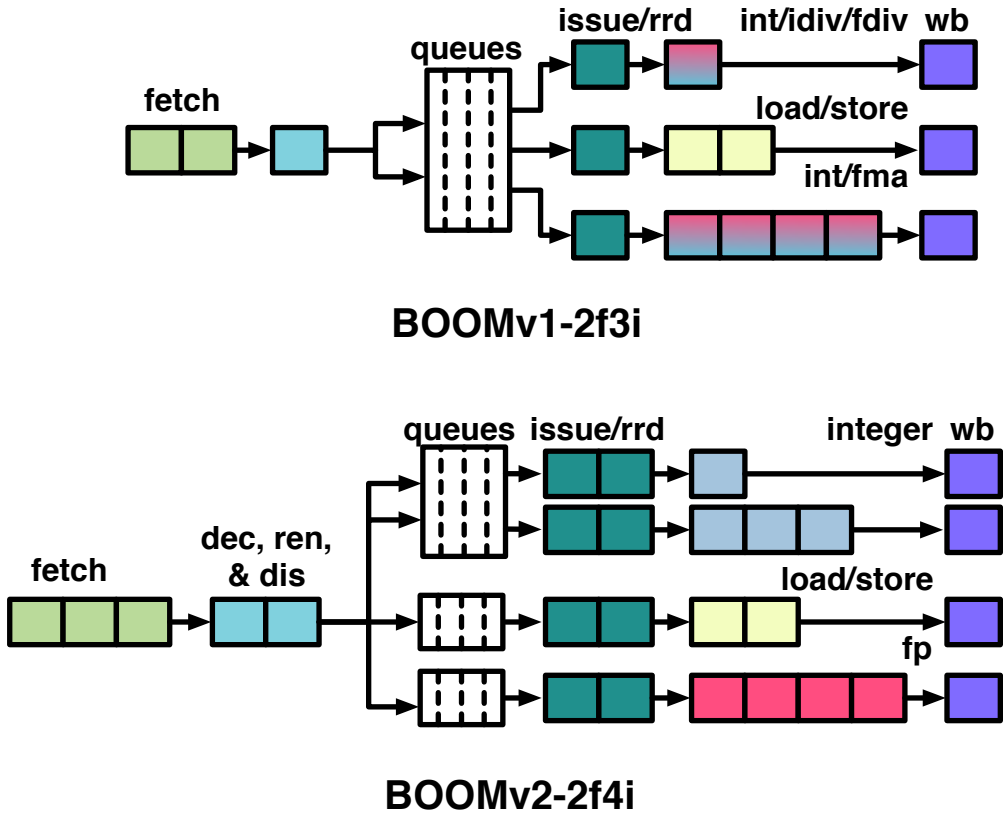


Figure 6.1: A comparison of a three-issue (*3i*) BOOMv1 and four-issue (*4i*) BOOMv2 pipeline both which can fetch two instructions every cycle (*2f*). Both show two integer ALU units, one memory unit, and one floating-point unit. Note that BOOMv2 uses a distributed issue window to reduce the issue port count for each separate issue window. In BOOMv1 the floating-point unit shares an issue port and register access ports with an integer ALU. Also note that BOOMv1 and BOOMv2 are parameterizable, allowing for wider issue widths than shown here.

Table 6.1: The parameters chosen for analysis of BOOM. Although BOOM is a parameterizable generator, for simplicity of discussion, we have limited our analysis to these two instantiations.

	BOOMv1	**BOOMv2**
BTB entries	40 (fully-associative)	64 x 4 (set-associative)
Fetch Width	2 insts	2 insts
Issue Width	3 micro-ops	4 micro-ops
Issue Entries	20	16/16/16
Regfile	7r3w	6r3w (int), 3r2w (fp)
Exe Units	iALU+iMul+FMA iALU+fDiv Load/Store	iALU+iMul+iDiv iALU FMA+fDiv Load/Store

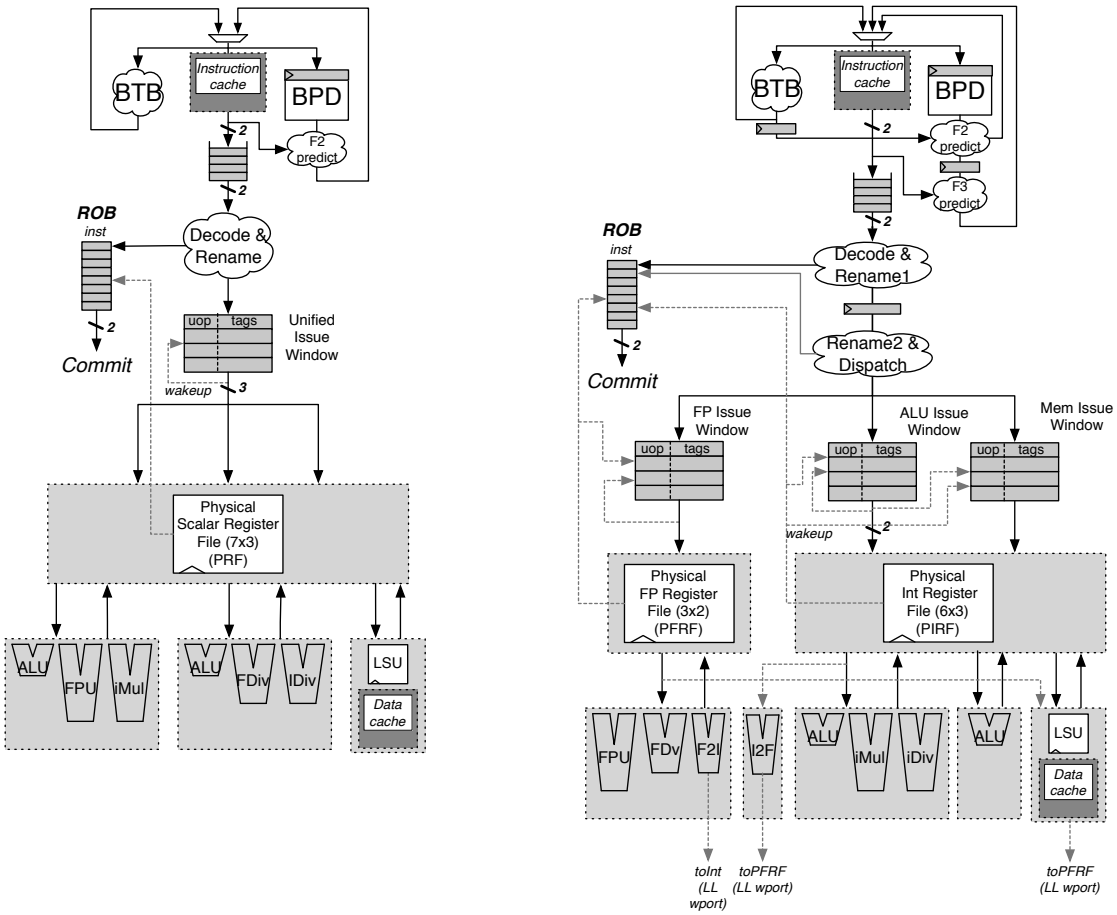
2-stage front-end pipeline design. Conditional branch prediction occurs after the branches have been decoded.

The design of BOOMv1 was partly informed by using educational technology libraries in conjunction with synthesis tools. While using educational libraries was useful for finding egregious mistakes in control logic signals, it was less useful in informing the organization of the datapaths. Most critically, we lacked access to a memory compiler. Although tools such as Cacti [122] can be used to analytically model the characteristics of memories, Cacti works best for reproducing memories that it has been tuned against such as single-port, cache-sized SRAMs. However, BOOM makes use of a multitude of smaller, irregular SRAMs for modules such as branch predictor tables, prediction snapshots, and address target buffers.

Upon analysis of the timing of BOOMv1 using TSMC 28 nm HPM, the following critical paths were identified:

1. issue window select
2. register rename busy-table read
3. conditional branch predictor redirect
4. register file read

The last path (register-read) only showed up as critical during post–place-and-route analysis.



(a) BOOMv1 (b) BOOMv2

Figure 6.2: The datapath changes between BOOMv1 and BOOMv2. Most notably, the issue window and physical register file have been distributed and an additional cycle has been added to the *Fetch* and *Rename* stages.

6.3 BOOMv2

BOOMv2 is an update to BOOMv1 based on information collected through synthesis, place, and route using a commercial TSMC 28 nm process. We performed the design space exploration by using standard single- and dual-ported memory compilers provided by the foundry, and by hand-crafting a standard-cell-based multi-ported register file. Figures 6.1 and 6.2 shows the organization of the BOOMv1 and BOOMv2 cores.

Work on BOOMv2 took place from April 9th through Aug 9th and included 4,948 additions and 2,377 deleted lines of code (LOC) out of the total 16k LOC code base. The following sections describe some of the major changes that comprise the BOOMv2 design.

6.3.1 Frontend (Instruction Fetch)

The purpose of the *frontend* is to fetch instructions for execution in the *backend*. Processor performance is best when the frontend provides an uninterrupted stream of instructions. This requires the frontend to utilize branch prediction techniques to predict which path it believes the instruction stream will take long before the branch can be properly resolved. Any mispredictions in the frontend will not be discovered until the branch (or jump-register) instruction is executed later in the backend. In the event of a misprediction, all instructions after the branch must be flushed from the processor and the frontend must be restarted using the correct instruction path.

The frontend end relies on a number of different branch prediction techniques to predict the instruction stream, each trading off accuracy, area, critical path cost, and pipeline penalty when making a prediction.

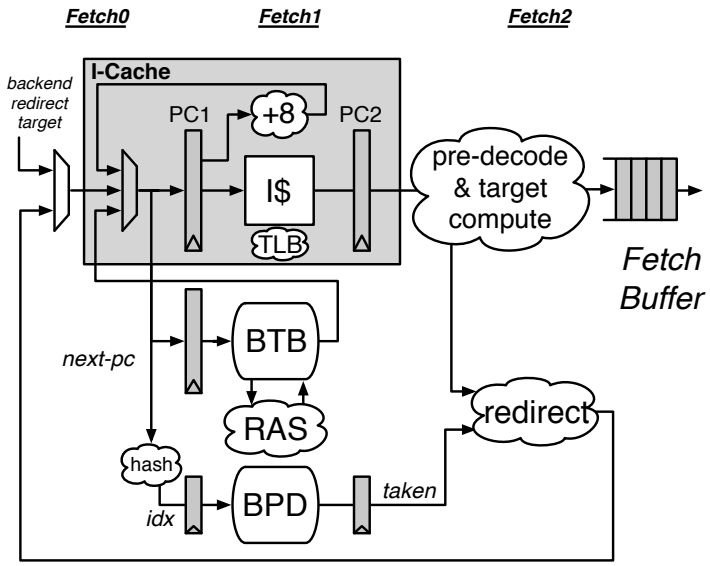
Branch Target Buffer (BTB) The BTB maintains a set of tables mapping from *instruction addresses* (PCs) to *branch targets*. When a lookup is performed, the *look-up address* indexes into the BTB and looks for any *tag matches*. If there is a *tag hit*, the BTB will make a prediction and may redirect the frontend based on its predicted *target address*. Some *hysteresis* bits are used to help guide the *taken/not-taken* decision of the BTB in the case of a *tag hit*. The BTB is a very expensive structure – for each BTB entry it must store the *tag* (anywhere from a partial tag of ≈20 bits to a full 64-bit tag[4]) and the *target* (a full 64 bit address[5]).

Return Address Stack (RAS) The RAS predicts function returns. *Jump-register* instructions are otherwise quite difficult to predict, as their target depends on a register value. However, functions are typically entered using a *Function Call* instruction at address A and return from the function using a *Return* instruction to address $A+1$.[6] – the RAS can detect the call, compute and then store the expected return address, and then later provide that

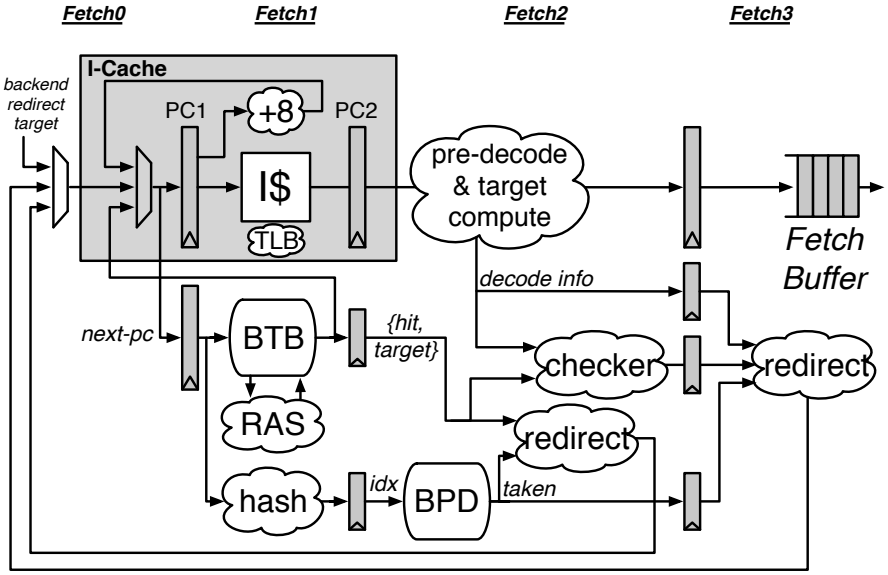
[4] Actually less than 64 bits since few 64-bit machines will support using all 64-bits as part of the virtual address.

[5] An offset from the look-up address could be used instead but that adds an *adder* to the critical path.

[6] Actually, it will be $A+4$ as the size of the call instruction to jump over is 4 bytes in RV64G.



(a) BOOMv1



(b) BOOMv2

Figure 6.3: The frontend pipelines for BOOMv1 and BOOMv2. To address critical path issues, BOOMv2 adds an extra stage. The branch predictor (BPD) index hashing function is moved to its own stage (F1), pushing back the BPD predictor table accesses a cycle as well (F2). BOOMv2 also utilizes a partially-tagged, set-associative BTB (BOOMv1 uses a fully-tagged fully-associative BTB). This requires adding a *checker* module which verifies that the predictions from the BTB matches the instructions being fetched.

predicted target when the *Return* is encountered. To support multiple nested function calls, the underlying RAS storage structure is a stack.

Conditional Branch Predictor (BPD). The BPD maintains a set of prediction and hysteresis tables to make *taken/not-taken* predictions based on a *look-up address*. The BPD *only* makes *taken/not-taken* predictions – it therefore relies on some other agent to provide information on what instructions are branches and what their targets are. The BPD can either use the BTB for this information or it can wait and decode the instructions themselves once they have been fetched from the instruction cache. Because the BPD does not store the expensive branch targets, it can be much denser and thus make more accurate predictions on the branch directions than the BTB – whereas each BTB entry may be 60 to 128 bits, the BPD may be as few as one or two bits per branch.[7] A common arch-type of BPD is a *global history* predictor. Global history predictors work by tracking the outcome of the last N branches in the program ("global") and hashing this *history* with the *look-up address* to compute a look-up index into the BPD prediction tables. For sophisticated BPD, this hashing function can become quite complex. BOOM's predictor tables are placed into single-ported SRAMs. Although many prediction tables are conceptually "tall and skinny" matrices (thousands of 2- or 4-bit entries), a generator written in *Chisel* transforms the predictor tables into a square memory structure to best match the SRAMs provided by a memory compiler.

Figure 6.3 shows the pipeline organization of the frontend. We found the a critical path in BOOMv1 to be the conditional branch predictor (BPD) making a prediction and redirecting the fetch instruction address in the F2 stage, as the BPD must first decode the newly fetched instructions and compute potential branch targets. For BOOMv2, we provide a full cycle to decode the instructions returning from the instruction cache and target computation (F2) and perform the redirection in the F3 stage. We also provide a full cycle for the hash indexing function, which removes the hashing off the critical path of *Next-PC selection*.

We have added the option for the BPD to continue to make predictions in F2 by using the BTB to provide the branch decode and target information. However, we found this path of accessing the prediction tables and redirecting the instruction stream in the same cycle to be too slow using the foundry-provided memory compilers.

Another critical path in the frontend was through the fully-associative, flip-flop-based BTB. We found roughly 40 entries to be the limit for a fully-associative BTB. We rewrote the BTB to be set-associative and designed to target single-ported memory. We experimented with placing the tags in flip-flop-based memories and in SRAM; the SRAM synthesized at a slower design point but place-and-routed better.

[7] We are ignoring the fact that predictors such as TAGE [89] actually do store partial tags which can allow them to predict which instructions are branches.

6.3.2 Distributed Issue Windows

The *issue window* holds all inflight and un-executed micro-ops (uops). Each *issue port* selects from one of the available *ready* uops to be issued. Some processors, such as Intel's Sandy Bridge processor, use a "unified reservation station" where all uops are placed in a single issue window. Other processors provide each functional unit its own issue window with a single issue select port. Each has its benefits and its challenges.

The size of the issue window denotes the number of in-flight, un-executed instructions that can be selected for out-of-order execution. The larger the window, the more instructions the scheduler can attempt to re-order. For BOOM, the issue window is implemented as a collapsing queue to allow the oldest instructions to be compressed towards the top. For issue-select, a cascading priority encoder selects the oldest instruction that is ready to issue. This path is exacerbated either by increasing the number of entries to search across or by increasing the number of issue ports. For BOOMv1, our synthesizable implementation of a 20 entry issue window with three issue ports was found to be too aggressive, so we switched to three distributed issue windows with 16 entries each (separate windows for integer, memory, and floating-point operations). This removes issue-select from the critical path while also increasing the total number of instructions that can be scheduled. However, to maintain performance of executing two integer ALU instructions and one memory instruction per cycle, a common configuration of BOOM will use two issue-select ports on the integer issue window.

6.3.3 Register File Design

One of the critical components of an out-of-order processor, and most resistant to synthesis efforts, is the multi-ported register file. As memory is expensive and time-consuming to access, modern processor architectures use *registers* to temporarily store their working set of data. These *registers* are aggregated into a *register file*. Instructions directly access the register file and send the data read out of the registers to the processor's functional units, whereby the resulting data is then *written back* to the register file. A modest processor that supports issuing simultaneously to two integer arithmetic units and a memory load/store unit requires 6 read ports and 3 write ports.

The register file in BOOMv1 provided many challenges – reading data out of the register file was a critical path, routing read data to functional units was a challenge for routing tools, and the register file itself failed to synthesize properly without failing the foundry design rules. Both the number of registers and the number of ports further exacerbate the challenges of synthesizing the register file.

We took two different approaches to improving the register file. The first level was purely microarchitectural. We split apart *issue-select* and *register-read* into two separate stages — *issue-select* is now given a full cycle to select and issue uops, and then another full cycle is given to read the operand data out of the register file. We lowered the register count by splitting up the unified physical register file into separate floating-point and integer register

files. This split also allowed us to reduce the read-port count by moving the three-operand fused-multiply add floating-point unit to the smaller floating-point register file.

The second path to improving the register file involved physical design. A significant problem in placing and routing a register file is the issue of *shorts* – a geometry violation in which two metal wires that should remain separate are physically attached. These shorts are caused by attempting to route too many wires to a relatively dense regfile array. BOOMv2's 70 entry integer register file of 6 read ports and 3 write ports comes to 4,480 bits, each needing 18 wires routed into and out of it. There is a mismatch between the synthesized array and the area needed to route all required wires, resulting in shorts. Although we were able to safely synthesize a 9 port register file in isolation with some guidance on the area placement and density targets, the routing tools were unable to physically place the synthesized register file without shorting wires when synthesizing the register file in place with the rest of the processor.

Instead, we opted to blackbox the *Chisel* register file and manually craft a register file *bit* out of foundry-provided standard cells. We then laid out each register bit in an array and let the placer automatically route the wires to and from each bit. While this fixed the *wire shorting* problem, the tri-state buffers struggled to drive each read wire across all 70 registers. We therefore implemented *hierarchical bitlines*; the bits are divided into clusters, tri-states drive the read ports inside of each cluster, and muxes select the read data across clusters.

As a counter-point, the smaller floating-point register file (three read ports, two write ports) is fully synthesized with no placement guidance.

6.4 Tapeout Methodology

Figure 6.5 shows all VLSI builds and their critical path lengths performed over a four month period as part of the BROOM tapeout effort. The reported clock frequency is calculated from the critical path, which is the summation of the target clock period and the negative clock skew. Data from post-synthesis ("syn") and post-place-and-route ("par") are shown and include builds performed at both the `slow-slow` ("SS") `typical-typical` ("TT") corners. Early builds were only of a BOOM core plus an L2 cache while later builds add in the resiliency ("res") hardware central to the main thesis of the BROOM chip. One should be careful of drawing conclusions from this figure; most builds resulted in LVS and DRC violations and many changes were made between each build. For example, early builds explored shrinking structure sizes to find the most fundamental critical paths while later builds sought to find the upper limits of structure sizing before the post-place-and-route critical path noticeably worsened. Table 6.2 provides a brief description of each VLSI build that is shown in Figure 6.5.

The BROOM tapeout effort started with a preliminary analysis of the BOOM's quality-of-result (QoR). This effort was performed using RVT-based cells and targeting the `TT` corner. By changing BOOM's configurations, we could build an intuition of what critical paths were

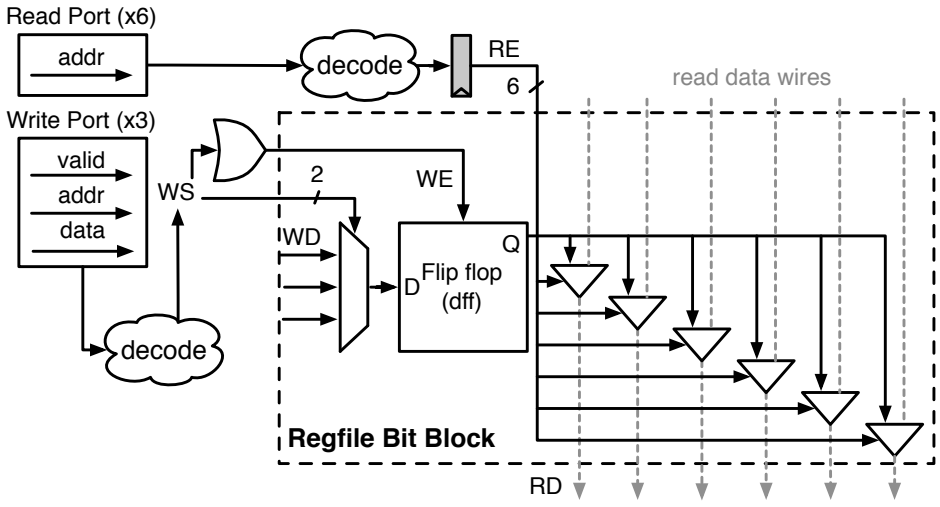


Figure 6.4: A Register File Bit manually crafted out of foundry-provided standard cells. Each read port provides a read-enable bit to signal a tri-state buffer to drive its port's read data line. The register file bits are laid out in an array for placement with guidance to the place tools. The tools are then allowed to automatically route the 18 wires into and out of each bit block.

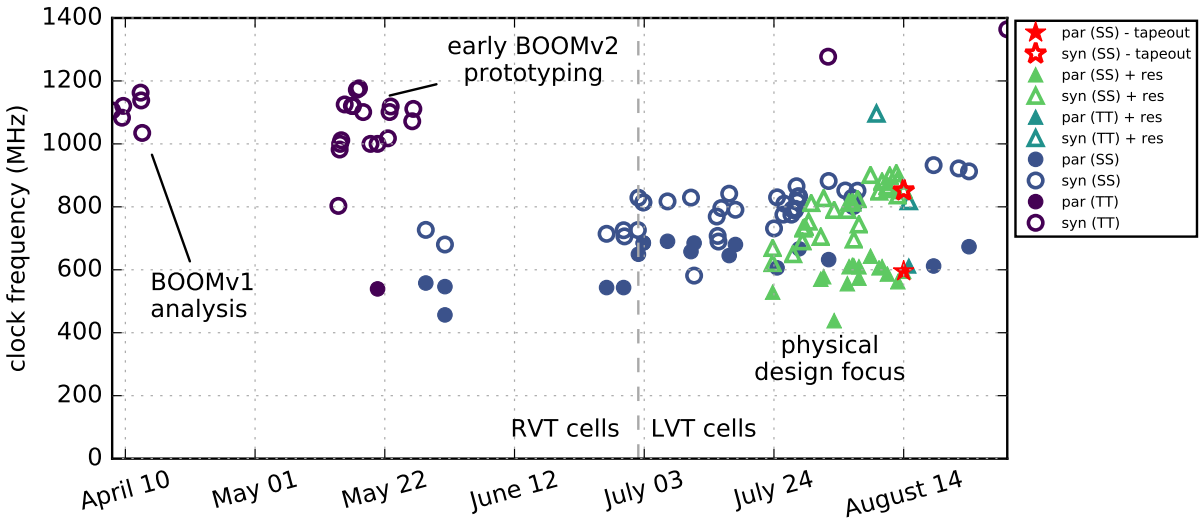


Figure 6.5: All VLSI builds are shown by date. Both `slow-slow (SS)` and `typical-typical (TT)` corners are shown. RVT cells were used initially but replaced with LVT cells starting in July. In the last month of the implementation effort, we added in the resiliency hardware (*"res"*) central to the research thesis of the chip which added to the critical path. While our design efforts slowly improved the post-synthesis critical paths, post-place-and-route reports showed the clock frequency was less amenable to our efforts.

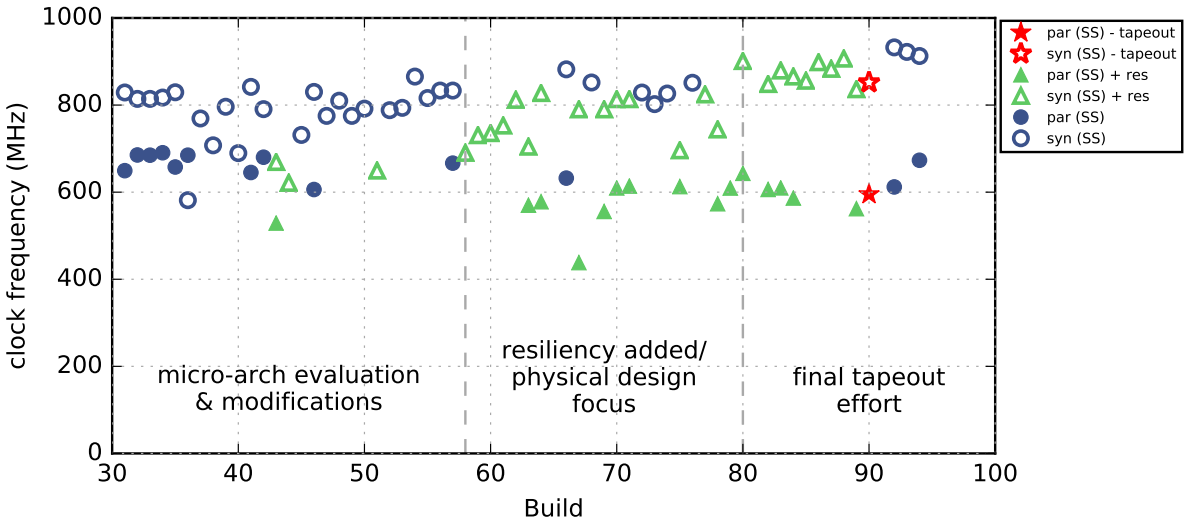


Figure 6.6: The `slow-slow` VLSI builds using LVT cells from July onwards are shown again and arranged by build number. Although the goal was a chip using resiliency techniques, we continued to build non-resilient designs to continue exploring critical paths in the BOOM core. Not shown is the impact of our design efforts on removing any LVS and DRC errors. Thus, many of the builds do *not* represent a manufacturable design.

truly critical and arrive at a plan of action for addressing these paths with a mixture of micro-architectural changes and physical design effort. For example, by removing an execution unit or shrinking the issue window size, we could better understand the benefits of design changes that would provide fewer issue ports per issue window. At this stage, we had concluded that four critical paths needed to be managed. As previously mentioned in Section 6.2, these critical paths were:

1. issue window select
2. register rename busy-table read
3. conditional branch predictor redirect
4. register file read

The micro-architectural changes to address the first two items together took one month. We also quickly prototyped a new frontend design that approximated a critical path fix for item three but was otherwise functionally incorrect. This frontend prototype helped justify the necessary design work before we committed to a full redesign of the frontend. We began testing these new changes in mid May and labeled the new design "BOOMv2". Figure 6.5 shows the cluster of activity that correspond to the BOOMv1 and early BOOMv2 analysis. After the initial BOOMv2 analysis was performed, another month of design effort went into BOOM to finish implementing the new frontend design and to apply changes based on the initial VLSI feedback.

Starting in late June, as the BOOMv2 RTL effort finished up, the implementation focus switched to physical design. Figure 6.6 shows the VLSI builds from July onwards that were performed using LVT-based cells at the `slow-slow` corner. Parameters in BOOM, for example the ROB size or the branch predictor sizing, were reduced to get a better feel for the fundamental critical paths that still required work and to find which modules had the greatest affect on DRC and LVS errors. At this stage, the clock frequency improved as the BOOM parameters were changed to instantiate a smaller BOOM core.

Once we were relatively happy with the BOOM micro-architecture, we added the resiliency hardware to the design. Many of these resiliency structures are on the critical paths of SRAM accesses. Thus, any VLSI builds with resiliency hardware enabled may generate analysis reports that hide critical paths that still need attention in the BOOM RTL. To allow improvements to both the resiliency structures and to the BOOM core to occur in parallel, we continued to perform VLSI builds with and without the resiliency hardware enabled (labeled "res" in Figures 6.5 and 6.6).

As our attention shifted to physical design issues, we began to explore a number of strategies for implementing a 6-read, 3-write register file. Although we attempted to use placement hints and floorplan scripts to help guide a synthesized register file design, we eventually settled on the semi-custom arrayed design discussed in Section 6.3.3 after coming

to the conclusion that the place and route tools could not manage a synthesized 9-port register file without committing geometry errors.

For the final stage of the implementation effort, we focused on fixing LVS and DRC errors while continuing to make small improvements to the critical paths that showed up in the place and route reports. We also began to increase structure sizes in BOOM that were no longer on the critical path in the post–place and route reports. For example, we quadrupled the size of the branch predictor.

Finally, on August 14th, we taped out an LVS clean, DRC sane design based on the submission deadline provided by TSMC. Up to (and past) that deadline, we continued making changes to the RTL to improve the QoR. Each additional build continued to provide us new critical paths to address. The final critical path of the place and routed design was through the resiliency error logging code.

More time would have allowed us to continue to improve both the clock frequency and the IPC performance of BROOM. With enough time available, we could have focused on fixing more systemic critical paths, particularly the write-back path for data returning from the data cache. Fixing this path would go hand-in-hand with fixing the IPC performance problem introduced by the longer load-to-use dependency. However, the rush to produce a LVS clean, DRC sane design that would be relatively free of RTL logic errors prevented us from committing to more risky and invasive fixes. Future VLSI implementation efforts can start from a known, good design point and can avoid the early exploratory builds that were needed for the BROOM tapeout.

6.5 Lessons Learned

The process of taking a register-transfer-level (RTL) design all the way through a modern VLSI tool flow has proven to be a very valuable experience.

Dealing with high port count memories and highly-congested wire routing are likely to require microarchitectural solutions. Dealing with the critical paths created by memories required microarchitectural changes that likely hurt IPC, which in turn motivates further microarchitectural changes. Lacking access to faithful memory models and layout early in the design process was a serious handicap. A manually-crafted cell approach is useful for exploring the design space before committing to a more custom design.

Memory timings are sensitive to their aspect ratios; tall, skinny memories do not work. We wrote *Chisel* generators to automatically translate large aspect ratio memories into rectangular structures by changing the index hashing functions and utilizing bit-masking of reads and writes.

Chasing down and fixing all critical paths can be a fool's errand. The most dominating critical path was the register file read as measured from post-place and route analysis. Fixing critical paths discovered from post-synthesis analysis may have only served to worsen IPC for little discernible gain in the final chip.

Table 6.2: The critical path length is reported from each of the VLSI builds during the BROOM tapeout. The critical path is the summation of the target clock period and the negative clock skew. Many changes were made beteween each run, only some of the major points are documented here. Run #90 is the final design that was submitted to TSMC for fabrication. Most of the other builds have violations.

Run	Corner	Resiliency	Description	Post-synthesis Cycle Time (ns)	Post-place-and-route Cycle Time (ns)
1	TT	-	first attempt using BOOMv1	0.904	
2	TT	-	retime idiv unit	0.923	
3	TT	-	retime idiv unit	0.892	
4	TT	-	reduce BTB entry count	0.860	
5	TT	-	disable BPD & fdivsqrt units	0.879	
6	TT	-	reduce issue window to 4 entries	0.966	
7	TT	-	begin using early prototype of BOOMv2	1.245	
8	TT	-		1.018	
9	TT	-	issue-width=1 (timing met)	1.000	
10	TT	-	reduce cycle time	0.989	
11	TT	-	fetch latency=3	0.889	
12	TT	-	fdivsqrt disabled	0.892	
13	TT	-	issue-width=2	0.893	
14	TT	-	integer issue window entries=10	0.853	
15	TT	-	decrease integer & fp issue window sizes	0.850	
16	TT	-	issue-width=2	0.908	
17	TT	-	1ns clock (syn timing met)	1.000	
18	TT	-	1ns clock (syn timing met)	1.000	1.855
19	TT	-	pipeline register read	0.983	
20	TT	-	add retiming to rename stage	0.908	
21	TT	-	retime & ungroup rename stage	0.894	
22	TT	-	rearrange constraint file	0.933	
23	TT	-	add compile_ultra -retime flag	0.900	
24	SS	-	move to SS corner	1.376	1.792
25	SS	-	Add resiliency	1.470	2.190
26	SS	-	fixes to floorplan		1.830
27	SS	-	multi-voltage domain	1.400	1.840
28	SS	-	loose cycle time	1.378	1.840
29	SS	-	loose cycle time with imul retiming	1.416	
30	SS	-	change int RF size to 70 registers	1.377	
31	SS	-	move to LVT std cells	1.206	1.540
32	SS	-	feed in floorplan to DC	1.228	1.459
33	SS	-	add -congestion in clock_opt_psyn flag	1.228	1.460
34	SS	-	change int RF size to 100 registers	1.224	1.448
35	SS	-	fixed top connection (passes RTL simulation)	1.206	1.520
36	SS	-	feed in new floorplan (2col4row for dcache)	1.720	1.460
37	SS	-	new set-associative BTB	1.300	
38	SS	-		1.413	
39	SS	-	retimed BPD pipeline and fetch unit w/o fp	1.256	
40	SS	-	with fp	1.450	
41	SS	-	use flip-flops for BTB	1.188	1.550
42	SS	-	use LVT SRAM for BTB	1.265	1.470
43	SS	yes	add in resiliency with BOOMv1	1.495	1.890
44	SS	yes	add in resisliency with BOOMv2	1.610	
45	SS	-	updated BTB w/ SRAM-based gshare BPD	1.367	

Run	Corner	Resiliency	Description	Post-synthesis Cycle Time (ns)	Post-place-and-route Cycle Time (ns)
46	SS	-	p table using FF; (BTB tag is also FF)	1.204	1.650
47	SS	-	try to remove a dependence on bpd request	1.290	
48	SS	-	RAS bypass calls = false	1.234	
49	SS	-	1.2 ns clock	1.290	
50	SS	-	1.1 ns clock	1.263	
51	SS	yes		1.540	
52	SS	-	1.0 ns clock	1.269	
53	SS	-	remove internal w→r bypass	1.260	
54	SS	-	with fp	1.155	
55	SS	-	RAS entries=0	1.225	
56	SS	-	RAS entries=0	1.200	
57	SS	-	RAS entries=8; with fp	1.200	1.500
58	SS	yes	remove I$ s1 disparity dependency	1.449	
59	SS	yes	with fp	1.369	
60	SS	yes	D$ bypass ecc (decode.uncorrected)	1.360	
61	SS	yes	I$ s1 tag disparity	1.328	
62	SS	yes	Set caches to ways=4; disable ECC	1.232	
63	SS	yes	use earlier commit	1.420	1.754
64	SS	yes	caches to ways=4; use SRAM for BPD and BTB	1.209	1.730
65	TT	-	TT corner	0.783	
66	SS	-	SS corner with new RTL commits	1.134	1.582
67	SS	yes	use new custom RF array; (logic error; coupled iss+rrd stages)	1.266	2.282
68	SS	-	with new RTL commits	1.174	
69	SS	yes	regfile array without muxing (with correct raddr register)	1.266	1.799
70	SS	yes	use hierarchical bitlines	1.230	1.640
71	SS	yes	use hierarchical bitlines; rerun previous	1.230	1.629
72	SS	-	add BPD prediction in F3	1.207	
73	SS	-	gshare history length increased to 13 bits	1.247	
74	SS	-	remove BPD F2 redirect	1.210	
75	SS	yes	add BPD prediction in F3; retime alu	1.437	1.632
76	SS	-	Fix correctness bug in BPD	1.175	
77	SS	yes	use flip-flops for BTB tags	1.213	
78	SS	yes	retime LSU; set history to 13b; BTB set to 2 ways	1.345	1.744
79	SS	yes			1.641
80	SS	yes	remove retiming on LSU; separate bb array; ECC logging	1.110	1.554
81	TT	yes		0.913	
82	SS	yes	halve gshare's htable; error logging changes	1.179	1.649
83	SS	yes	fix LSU	1.137	1.640
84	SS	yes	ECC logging fixes	1.156	1.705
85	SS	yes	fix critical path on valid bit	1.169	
86	SS	yes	add L2 miss counter	1.113	
87	SS	yes	BTB update	1.133	
88	SS	yes	delay write to fp regfile	1.103	
89	SS	yes	fix L1 I$ bb_array connection	1.196	1.779
90	SS	yes	final tapeout version	1.174	1.680
91	TT	yes	after tapeout fixes; remove timing analysis for ECC logging; fix placement in L2 tags	1.223	1.633
92	SS	-	delay write to fp RF; use flip-flops for BTB tags; use synthesized int regfile	1.072	1.634
93	SS	-	use SRAM for BTB tags; use custom int regfile	1.085	
94	SS	-	set_dont_touch to the regfile	1.096	1.485
95	TT	-	TT corner	0.733	

Describing hardware using generators proved to be a very useful technique; multiple design points could be generated and evaluated, and the final design choices could be committed to later in the design cycle. We could also increase our confidence that particular critical paths were worth pursuing; by removing functional units and register read ports, we could estimate the improvement from microarchitectural techniques that would reduce port counts on the issue windows and register file.

Chisel is a wonderfully expressive language. With a proper software engineering of the code base, radical changes to the datapaths can be made very quickly. Splitting up the register files and issue windows was a one week effort, and pipelining the *register-rename* stage was another week. However, physical design is a stumbling block to agile hardware development. Small changes could be reasoned about and executed swiftly, but larger changes could change the physical layout of the chip and dramatically affect critical paths and the associated costs of the new design point.

6.6 What Does It Take To Go Really Fast?

A number of challenges exist to push BOOM below 35 FO4. First the L1 instruction and data caches would need to be redesigned. Both caches return data after a single cycle (they can maintain a throughput of one request a cycle to any address that hits in the cache). This path is roughly 35 FO4. A few techniques exist to increase clock frequency but increase the latency of cache accesses.

For this analysis, we used regular threshold voltage (RVT)-based SRAM. However, the BTB is a crucial performance structure typically designed to be accessed and used to make a prediction within a single-cycle and is thus a prime suspect for additional custom effort. Another solution is to increase the latency of BTB predictions and accept a bubble inserted into the fetch pipeline on *taken* branch predictions.

There are many other structures that are often the focus of manual attention: functional units; content-addressable memories (CAMs) are crucial for many structures in out-of-order processors such as the load/store unit or translation lookaside buffers (TLBs) [47]; and the issue-select logic can dictate how large of an issue window can be deployed and ultimately guide how many instructions can be inflight in an out-of-order processor.

However, any techniques to increase BOOM's clock frequency will have to be balanced against decreasing the IPC performance. For example, BOOM's new front-end suffers from additional bubbles even on correct branch predictions. Additional strategies will need to be employed to remove these bubbles when predictors are predicting correctly [94].

6.7 Conclusion

Modern out-of-order processors rely on a number of memory macros and arrays of different shapes and sizes, and many of them appear in the critical path. The impact on the ac-

tual critical path is hard to assess by using flip-flop-based arrays and academic/educational modeling tools, because they may either yield physically unimplementable designs or generate designs with poor performance and power characteristics. Re-architecting the design by relying on a hand-crafted, yet synthesizable register file array and leveraging hardware generators written in *Chisel* helped us isolate real critical paths from false ones. This methodology narrows down the range of arrays that would eventually have to be handcrafted for a serious production-quality implementation.

As BOOMv2 has largely been an effort in improving the critical path of BOOM, there has been an expected drop in Instruction per cycle (IPC) performance. Using the Coremark benchmark, we witnessed up to a 20% drop in IPC based on the parameters listed in Table 6.1. Over half of this performance degradation is due to the increased latency between load instructions and any dependent instructions. There are a number of available techniques to address this that BOOM does not currently employ. However, BOOMv2 adds a number of parameter options that allows the designer to configure the pipeline depth of the register renaming and register-read components, allowing BOOMv2 to recapture most of the lost IPC in exchange for an increased clock period.

Chapter 7

Conclusion

Details matter. Continued innovation in the processor space must come from performance improvements that are power-neutral, or power improvements that increase the headroom to allow for more aggressive performance techniques. Building real systems not only provides a stable basis from which to perform detailed performance and power analyses, but also provides deeper insight and intuition that is molded by exposure to both low-level details and cross-cutting issues. As one example, in BOOM the interplay between the issue queue select ports, the register file read ports, and large-scale memory array process technology drove us to re-evaluate and re-implement multiple parts of BOOM once we had access to a real commercial tool-flow. Mimicking designs from older generations — in which different implementation methodologies and logic styles were used — left us under-informed compared to the knowledge we gained in analyzing our own designs using a modern tool-flow.

Some of the most exciting research has been built upon the works of others. As recent examples, the *Rocket-chip* SoC has shown up in the first silicon-photonics-enabled microprocessor (MIT, CU Boulder, Berkeley) [100] and in the 511-core *Celerity* processor (UCSD, Cornell, Michigan, and UCLA) [4]. Similarly, the open-source UltraSPARC T1 processor from Sun has been used to help build the 25-core *OpenPiton* processor (Princeton) [11], and the UltraSPARC T2 was used as part of the heterogeneous *Fabscalar* chip (NCSU) [84].

Likewise, this work has been in part made possible by leveraging existing infrastructure newly available in the open-source hardware ecosystem; namely, the RISC-V Instruction Set Architecture, the *Chisel* hardware construction language, and the *Rocket-chip* SoC generator.

Hopefully, BOOM can can serve as a platform for the next set of implementers and researchers. By providing a detailed implementation of a high-performance, general-purpose, open-source processor, we hope to enable researchers to pursue new avenues of study that require full systems that can provide a higher fidelity of results. And hopefully, they too can learn from it as much as I have.

7.1 Contributions

This thesis includes the following contributions:

- **A complete implementation of a superscalar, out-of-order processor generator** — We built a superscalar, out-of-order processor generator called BOOM that is capable of booting the Linux operating system and running user-level applications. Many of the details in implementing a full system kept us honest. The floating-point support is often a missing piece in academic projects, but it served to motivate many of the challenges and learning experiences in building a parameterizable generator capable of running general-purpose codes. Many corners can be cut in integer-only cores, but floating-point instructions begin to stretch the boundaries of "RISC" ISAs. These extra challenges include three register operands, longer instruction latencies, implicit condition codes, condition flag side-effects, and register moves between a new set of architectural registers.

- **A competitive implementation** — BOOM achieves comparable (or better) branch prediction accuracy and instructions-per-cycle performance relative to similarly-sized industry out-of-order processors. The pursuit for a competitive processor led us to focus on new areas we would have otherwise avoided. We needed to support real benchmarks, which forced us to support floating-point instructions and atomics. We also spend considerable time on branch prediction — without a competitive branch predictor, no other improvements would have mattered. And unfortunately, if a research processor significantly lags in performance to commercial offerings, many studies using it become less informative.

- **A productive implementation** — We demonstrated our productive processor generator design by implementing it using only 16k lines of code. In order for others to build off of our work, our platform must be easy to understand, easy to modify, and easy to extend.

- **Demonstrated productivity with an agile tape-out** — We further demonstrated our productivity and agility by making significant micro-architectural design changes as part of a two-person tape-out performed over four months. We demonstrated that it is possible to take the BOOM *Chisel* design to silicon, and that *Chisel* does not preclude lower-level physical design optimizations. We showed both what kind of changes are possible to be performed quickly using *Chisel*, as well as showed what physical design hurdles still remain. Hopefully, this effort paves the way for others to take BOOM to silicon too.

7.2 Future Directions

As BOOM is an open-source processor generator, there is a opportunity for others to take BOOM in new directions, to use it to augment their own research, or to simply improve and contribute back to BOOM. This section lists some of the opportunities and interesting questions that BOOM can help explore.

Implementing new RISC-V instruction extensions BOOM is a scalar processor that implements the "G" general-purpose subset of the RISC-V ISA. However, the RISC-V Foundation is actively developing new extensions for RISC-V, some that may radically alter the micro-architecture of future "general-purpose" processors. For example, all contemporary high-performance, general-purpose commercial processors implement some form of data-parallel acceleration, typically in the form of SIMD (Single Instruction Multiple Data), which allows one instruction to operate on multiple elements in parallel. Even for general-purpose codes, SIMD instructions can be used to accelerate the zeroing and copying of large arrays in memory. The RISC-V Foundation is developing a Vector Extension that will provide the same benefits enjoyed by other ISAs such as the x86 SSE and AVX SIMD extensions and ARM's NEON extension. However, the current RISC-V Vector Extension proposal provides new opportunities and complexity challenges beyond SSE, AVX, and NEON. Some of these challenges include its reconfigurable vector length, its polymorphic encoding, its usage of a vector data register as an implicit predicate register, and its reconfigurable scalar, vector, and matrix register *shapes*. How best to address these challenges and map a competitive design to an out-of-order pipeline is an interesting challenge. Other extensions have been proposed, some which may map poorly to an out-of-order pipeline. Some of these proposed instructions include adding loop-count instructions, overflow arithmetic, and integer-packed SIMD to help support applications such as specialized digital signal processing (DSP), managed run-time languages, and low-power energy-efficient computing.

Co-processor and accelerator interfaces The *Rocket* in-order processor has found significant success in part by providing a co-processor/accelerator interface called RoCC. Example accelerators that have used the RoCC interface include the *Hwacha* vector-thread processor [106], the *DANA* neural network accelerator [30], and the *Celerity* binarized neural network (BNN) specialized accelerator [4]. Implementing the RoCC interface in BOOM would provide researchers a higher-performance control processor for their research. It would also open up new and interesting research questions regarding the interactions between out-of-order cores and their co-processors. For example, what would be the performance benefits of implementing the RISC-V Vector Extension as a RoCC co-processor (in which all RoCC commands are sent at instruction commit) versus a more tightly integrated vector unit within BOOM that exercises register renaming and speculative out-of-order vector execution?

Design verification *Chisel* has facilitated rapid and agile RTL implementation efforts. However, design verification is still a significant obstacle and the best approaches to reduce

the verification load is still an open question. As a complex IP block, BOOM can serve as a platform of study for future verification efforts. As an example of such early stage efforts, the most recent bug found in BOOM manifested at over 400 billion cycles into simulation. Finding this bug required a sophisticated FPGA-accelerated simulation framework, built upon the Strober infrastructure [56], that used synthesizable assertions to find the error and a state snapshot-and-replay capability to extract a waveform from the failing scenario [8].

Agile physical design Similar to design verification, physical design is also a significant hurdle to cheaper, more agile hardware development. Although *Chisel* allows for RTL changes to be made quickly, some RTL changes can have significant effects on the physical design. For example, an added memory block can take up too much area and cause routing congestion. Human intervention may be needed to guide changes to the floor-plan to alleviate the new routing congestion, or worse, a micro-architectural change at the RTL level may be necessary to reduce timing pressure. Large-scale memory arrays pose another problem for agile physical design. There remains significant value in crafting custom memory arrays over foundry-provided memory compilers, which unfortunately can require designers to make binding decisions earlier in the design process than they would normally prefer.

Design space exploration (DSE) A challenge with hardware generators is how does the designer choose a specific parameter set to instantiate in silicon? A design like BOOM can provide enough configuration options to create a design space of millions of unique designs. As each instance requires hours to evaluate, and CAD tool-flows are too expensive to allow for significant parallelism of this analysis, care must be taken to decide which design points are worth pursuing. Ideally, an intelligent DSE search algorithm would only choose points that were likely to appear near the pareto-optimal frontier.

Performance improvements With five graduate-student-years of design effort, BOOM achieves a modest level of performance that is on par with some of the older out-of-order designs that are in ARM's current IP catalogue. However, there is significant head-room in improving upon BOOM's performance that can be explored. A couple areas for performance improvement include data-prefetching, larger issue queue designs, memory disambiguator predictors, accelerated atomic and fence instruction execution, a wider superscalar width, and a more aggressive instruction fetch unit.

7.3 Final Remarks

The end of Moore's Law and Dennard Scaling brings us into a period that is both scary and exciting. Manufacturing process technology alone can no longer give us better chips. Instead, the onus is on the architects and the micro-architects to continue taking progress to new heights.

Architects have a lot to be thankful for. Silicon transistors are incredibly small, fast, plentiful, and cheap. Commodity chips routinely ship with hundreds of millions, even billions, of transistors running at a few gigahertz with air-cooling. New fields are growing into billion dollar industries: search, mobile, automative, and internet-of-things demand increased focus on the topics of machine learning, inference, sensing, and processing. These new applications are motivating new innovations in architecture and micro-architecture, demanding novel and highly-specialized accelerators to extract the best performance for a given power envelope.

One of the challenging aspects of a more specialized future is that chip companies must cope with concurrently shipping more and increasingly specialized designs, while also shipping fewer units of any particular design. This diminishing ability to amortize the design and verification costs requires design teams to achieve more with fewer resources.

This suggests a future of hardware design that will likely focus on enabling implementers to be more agile and more able to cheaply explore new designs. However, implementation is not the only challenging part to chip design: physical design and verification will need to undergo similar revolutions in agile methodology.

FPGAs have gotten small enough to provide a relatively cheap option for companies and hobbyists to prototype hardware designs, reducing some of the risk in taking designs to silicon. This option has also opened up the door for creators to share and use hardware IP without the expenses of silicon fabrication, ushering in a new period of open-source hardware.

The slowing down of Moore's Law and the longer process node periods also provide another hidden benefit: slow and steady can win the race. Previously, process iterations were so quick that designs had to begin conception well in advance of that particular process becoming commoditized. The period between design conception and product delivery had to be short enough to avoid targeting a process that became obsolete by a product's launch [48]. With more time between process switches, more knowledge can be built up on each node. Designs also improve more slowly, as they must rely on micro-architectural innovation to realize continued performance improvements. It is now more possible than ever for academics and hobbyists to catch up to industry designs.

The out-of-order core is just one piece of the future computing landscape. While well-suited for general-purpose applications, out-of-order cores will need to interface with specialized accelerators and devices to deliver the best use of the limited transistor- and power-budgets.

Hopefully, BOOM can play a small, but important role in this future.

Appendix A

A Selection of Encountered Bugs

Possibly the biggest gamble that was made in the course of this thesis was choosing a project that had to actually function correctly! In most micro-architectural software simulator implementations, the functional model and the timing model are separated such that any errors in the timing model — where the research ideas are implemented — do not affect the correctness of the workloads being analyzed. But as BOOM *is* a processor, there is no such functional/timing model split. Thus, if the implementation of a research idea affects the technical correctness of BOOM, then the benchmarks will likely crash. This is not conducive to productive research! In that spirit, this section shares just a few of the bugs that were encountered during the development of BOOM. At a minimum, we hope that this section can encourage and inspire work in validation and testing.

The following bugs cover a range of categories and are ordered roughly in the chronological order that they were found. Some of these bugs were found quickly after implementing new features, while other bugs lingered and took a fine-tooth comb to suss out. Some were the fault of typographical or copy/paste mistakes. Some were bugs in other libraries — a few were found in the `berkeley-hardfloat` units and one was encountered in *Chisel* 1.0's simulation library. One of the more common classes of bug in BOOM involved the failure to properly kill a misspeculated instruction, which was often due to failing to update an instruction's *branch-mask* properly or failing to check if the instruction should be killed. These *suppose-to-be-killed* instructions would often bounce around the processor and wreak havoc on the micro-architectural state. Another common class of errors was caused by changing interfaces in which signals could be left dangling or in which contracts between modules changed, particularly with interfaces to external projects like *Rocket-chip*.[1]

Finding many of these bugs has been slow and tedious. Some bugs were found using the `riscv-torture` fuzzer [82] while others required using assertions to catch them. Performance bugs were especially difficult to track down, but thankfully are less important than correctness bugs. The worst bugs to solve are those that only arise when executing long-running applications that simply hang! A number of techniques were used to find the

[1]The latest *Chisel* 3.0.0 provides stronger protections against dangling I/O signals.

source of these bugs. Assertions were the favored method, but the comparison of commit logs, waveforms, and even printfs were heavily utilized. Having an existing processor in *Rocket* to compare against often proved invaluable. The `spike` ISA simulator was another valuable tool, however, it proved to be surprisingly difficult to get `spike` and BOOM to agree on the trace of committed instructions exactly. Many valid differences made it difficult to compare BOOM to `spike`, including the load-reserve/store-conditional instruction, timer interrupts, external stimuli, and platform deviations. Recent research has been exploring FPGA-accelerated validation as a new tool for validating *Chisel*-based designs executing very long workloads [8].

Surprisingly poor performance exhibited — An early implementation of a two-wide superscalar issue design was causing any given ALU operation to be issued to both ALU functional units. This mistake turned a dual-issue machine into a very expensive single-issue machine. The bug was caused by the first issue selector not properly informing the second issue selector that it had chosen that particular instruction already.

Multiple branches sharing the same branch tag — A full branch mask was not properly stalling the *Decode* stage until more branch snapshots could be allocated. This bug caused new branches to be allocated old (but still in-use) branch tags and chaos to break loose when one of the newer branches misspeculated and killed instructions older than itself. This error was caused by a typo: `dis_mask` (dispatch-stage valid mask) was being used instead of `dec_mask` (decode-stage valid mask), which corresponds to the valid bits in a different stage in the processor.

Failing simple arithmetic tests — Not having enough bits specified in the width of the ALU control signals caused the `ListLookup` decode table to generate incorrect logic.

Fetching incorrect instruction bits — The instruction cache improperly latched a request when suffering from a cache miss. This error was only exercised under two conditions: 1) when the data cache also missed in the cache causing back-pressure to the instruction cache as only one request is allowed to go off-tile per cycle, and 2) when a branch misspeculation occurred on the next cycle causing a new and different instruction to be requested. The original instruction bits eventually returned but were given the new instruction's tag.

Some instructions got skipped and were never executed after an exception — The ROB can commit multiple instructions per cycle. In this case, the ROB sees an exception in the *commit bundle* and takes it, but fails to notice the *busy* instructions that are older than the excepting instruction that must be allowed to finish first. These inflight instructions would get lost as the exception is taken too early.

The exception pc was being set improperly — On an exception, the `epc` was set to the aligned pc, and not the pc of the excepting instruction. This bug was hard to notice

because 1) often the aligned pc matched the excepting pc, and 2) the instructions starting at the aligned pc were typically idempotent.

The wrong store data could get bypassed to the dependent load — If two stores at the same address existed in the Store Queue, it was possible for the load to get data from the wrong store. This error was typographic: the `sdq_val` signal ("is the store data valid?") was used to help generate the address-match bit-vector instead of the `saq_val` signal ("is the store address valid?").

Load misspeculation error — A store checks the Load Address Queue (LAQ) for any younger loads that have incorrectly executed ahead of the store. If multiple loads fail, only the oldest load needs to be tracked since its pipeline flush will dominate the younger load. However, in this error, a fixed priority encoder was used instead of an age-based priority encoder, causing the wrong load to be marked as having failed. The older load would then proceed with having incorrect data and the younger load would be replayed. A commit log comparison against an ISA simulator allowed us to find the specific load that was receiving stale data.

Iterative muldiv unit writing bad data to the register file — If the iterative muldiv unit received a request on the same cycle that the request was killed, the request incorrectly continued executing and would eventually write back to a physical register. However, as the iterative unit can take tens of cycles to resolve, it was likely that that particular physical register would be added to the freelist and then reallocated to a new instruction. This new instruction would then receive a garbage value.

Incorrect value from muldiv unit — When dividing by zero, the integer muldiv unit incorrectly returned the absolute value of x instead of x.

An ECALL instruction crashes the C++ simulator process — A *fetch packet* contains a (branch, ecall) pair. The Decode Stage does not properly stall the ECALL instruction until the branch direction is resolved. Although the branch is suppose to direct the instruction stream around the ECALL instruction, the ECALL instruction executes a bad syscall number, an invalid address access is attempted in the host memory, and the simulator process crashes.

401.bzip suffers a hanged pipeline 33 million cycles in — A branch is misspeculated, but a load on this misspeculated path still writes back data to the register file and clears its ROB busy-bit. This problematic load had been sleeping in the LAQ and retried on the same cycle as the branch was resolved. This load fails to get killed and instead returns its data from a matching (and non-speculative) store. Had the load tried to get its data from the data cache it would have properly been killed. The error was caused by reading the wrong stage's `br-mask`. The correct signal was `l_uop.br_mask`, instead of `r_mem_uop.br_mask`.

Rocket BTB bug leads to incorrect instruction PC trace — The BTB was predicting a *taken* branch as jumping incorrectly to PC+4 instead of its proper target. In actuality, the BTB should have either predicted *not-taken* or predicted *taken* but with the correct target. The *Rocket* in-order core only checks the expected target matches the actual target and was fine with this turn of events of a "taken" branch to PC+4. BOOM was checking only the branch direction, saw *taken*, and decided the branch had been correctly speculated.

Trying to run Linux for the first time; throws misaligned fetch exception — A JALR instruction jumps to the wrong address. Linux was the first program BOOM ran that is placed in the "negative" addresses. The error was a lack of sign-extension of the PC when used as an operand for the AUIPC instruction.

Linux hangs; spinning on an exception to an unknown instruction — A timer interrupt incorrectly jumps to a faulting page when attempting to execute the interrupt handler. The PC for `evec` (exception vector) was not being properly sign-extended.

The dhrystone benchmark was printing out "%" instead of carriage returns — The BTB predicted *not-taken* for a *fetch packet* with (branch, jump). The frontend failed to redirect the unconditional jump, as it decided to not override the BTB's prediction of the branch that preceded it.

Increasing the BTB beyond 64 entries causes target program crashes — The BTB's associative search generates a one-hot encoded bit vector. A bug in early *Chisel* 1.0's C++ multiword simulation library incorrectly handled one-hot encodings that were larger than the native register width of the host machine (64-bits in this case).

BOOM hangs; AMO never succeeds — The Inflight Load Queue (IFLQ) was leaking entries and eventually filled up completely, preventing forward progress. The problem was that when the data cache nacked an AMO, the AMO's entry in the IFLQ was not cleared as it was not a "load".

BOOM believes that $+0.0$ and -0.0 are different — Multiplying a negative 0.0 could cause the raw bits in the 65-bit recoded floating-point format to mismatch against an unscaled 0.0. This was a bug in the `berkeley-hardfloat` comparison units. Similarly, infinities could also be scaled and caused to erroneously mismatch too. Although these errors were found using torture, which generates a small test program when a failure is detected, this error was annoyingly difficult to reproduce since smaller snippets from the failing torture test failed to reproduce the error. In order to reproduce the error, the test program needed to multiply using a number like 3 that would change the mantissa.

Unreasonably bad performance when load misspeculations occurred — When running large programs with a non-zero number of store-load ordering misspeculations, performance was significantly worse than expected. It was noticed that branch prediction accuracy correlated strongly with the frequency of store-load ordering misspeculations. Each ordering failure causes a pipeline flush and retry. The eventual culprit was the global history of the branch predictor was not being properly rolled-back when taking these "mini-exceptions." The fix was to maintain a commit check-point copy of the global history.

Poor branch prediction performance in a hand-crafted test — A small "branch obstacle course" returned unexpectedly poor performance. A bad indexing math error caused the most significant bit of the global branch history to be ignored.

The `berkeley-hardfloat` floating-point square-root unit returned the wrong answer for the input value 171 — This was found with overnight torture testing.

BOOM ran fine in Verilator but hanged on the FPGA — A new signal was added to a top-level I/O in the *Rocket-chip* tile which drives the *reset-vector* (which address do we start executing from when the core comes out of reset?). In Verilator simulation, the Debug Transport Module immediately forces BOOM to begin loading the test binary by executing a program out of the Program Buffer. However, the FPGA platform used a faster Tethered Serial Interface (TSI) to load the program while BOOM is held in reset. Once finished, the TSI then removes BOOM from reset and the TSI-specific boot-loader has BOOM jump to the application code. But since BOOM begins incorrectly at a random address after reset instead of in the boot-loader, the FPGA emulation hanged.

BOOM hangs 50% of the time running complex applications after updating *Rocket-chip* — A refactoring of the I/O interfaces left the invalidate-load-reservation signal dangling. Thus, for half of the simulation executions the signal was randomized to 1, or "always-invalidate-load-reservation", which prevents forward progress for any binary that uses load-reserve instructions.

The 445.gobmk benchmark returns an assertion error at 14.9 billion cycles — Using an FPGA simulation with synthesized assertions, we found a bug in which a misspeculated FP-to-Int move instruction incorrectly writes back to an invalid ROB entry. The FP-to-Int instruction suffers a structural hazard in accessing the write-back port on the register file due to a load instruction. During this same cycle a branch misprediction kills all the entries in the FP-to-Int queue. A copy/paste bug from a previous flow-through queue implementation causes `enq.valid` to set `deq.valid` to true even though the `enq.bits` is reading from a misspeculated entry before the dequeue pointer has been updated. Without FPGA acceleration, this bug would have required 431 days of Verilator runtime.

The 401.bzip2 benchmark returns an assertion error at 500 billion cycles — A JAL instruction jumps to the wrong target. This is caused by improper signed arithmetic in the frontend's handling of the unconditional jump target. This was found using FPGA-accelerated validation. Otherwise, this would have required 39 years of Verilator simulation to find.

A kernel panic occurs on an illegal instruction when reading the `rdtime` CSR — The latest RISC-V Privileged Specification v1.10 adds a feature to support faster instruction emulation by writing the illegal instruction bits into the control-status register (CSR) `mtval` (previously `mbadaddr`). BOOM instead wrote the excepting PC into `mtval`, continuing with the previous v1.9 behavior for `mbadaddr`. A read of the CSR `rdtime` is suppose to throw an illegal instruction on the *Rocket-chip* platform so that the machine mode can then emulate the CSR read by accessing the memory-mapped real-time clock and returning that value instead. However, by writing the PC into `mtval`, the emulation routine was confused and thought the illegal instruction was in fact an illegal instruction instead of a read of the `rdtime` CSR. This bug required a comparison of *Rocket*'s commit log and waveform against BOOM's commit log and waveform to track the correct instruction path through the different privilege levels. Part of the fix involved patching the upstream RISC-V conformance tests to check for proper illegal instruction handling.

Bibliography

[1] Jenny C Aker and Isaac M Mbiti. "Mobile phones and economic development in Africa". *The Journal of Economic Perspectives* 24.3 (2010), pp. 207–232.

[2] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[3] Mark Anderson. "A more cerebral cortex". *IEEE Spectrum* 47.1 (2010).

[4] Tutu Ajayi Khalid Al-Hawaj Aporva, Amarnath Steve Dai Scott Davidson, Paul Gao Gai Liu Anuj Rao, Austin Rovinski Ningxiao Sun Christopher Torng, Luis Vega Bandhav Veluri Shaolin Xie, and Chun Zhao Ritchie Zhao. "Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm" (2017).

[5] Ehsan K Ardestani and Jose Renau. "ESESC: A fast multicore simulator using time-based sampling". *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*. IEEE. 2013, pp. 448–459.

[6] *ARM Outmuscles Atom on Benchmark*. `http://parisbocek.typepad.com/blog/2011/04/arm-outmuscles-atom-on-benchmark-1.html/`.

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

[8] Krste Asanović, Jonathan Bachrach, David Biancolin, Christopher Celio, Sagar Karandikar, and Donggyu Kim. "Deterministic FPGA-Accelerated RTL Validation with Replayable Errors". *[In submission]*.

[9] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014.

[10] W. Ashmawi, J. Burr, A. Sharma, and J. Renau. "Implementation of a Power Efficient High Performance FPU for SCOORE". *Workshop on Architectural Research Prototyping (WARP), held in conjunction with ISCA-35*. 2008.

[11] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. "OpenPiton: An open source manycore research framework". *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 217–232.

[12] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[13] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. "Full-system chip multiprocessor power evaluations using FPGA-based emulation". *Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*. IEEE. 2008, pp. 335–340.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. "The GEM5 Simulator". *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*. Vol. 28. 2. ACM, 2000.

[16] Doug Burger and Todd M Austin. "The SimpleScalar tool set, version 2.0". *ACM SIGARCH computer architecture news* 25.3 (1997), pp. 13–25.

[17] Doug Burger, Todd M Austin, and Steve Bennett. *Evaluating future microprocessors: The simplescalar tool set*. University of Wisconsin-Madison, Computer Sciences Department, 1996.

[18] Brad Burgess. "Samsung Exynos M1 Processor". *Hot Chips*. Samsung Austin R&D Center – SARC. 2016.

[19] Cadence Design Systems. *Palladium Accelerator/Emulator*. `http://www.cadence.com/products/functional_ver/palladium/`.

[20] *Cadence Palladium XP II Verication Computing Platform*. Tech. rep. Cadence Design Systems, 2013.

[21] *Championship Branch Prediction (CBP-5)*. `https://www.jilp.org/cbp2016/`.

[22] David G Chinnery and Kurt Keutzer. "Closing the power gap between ASIC and custom: an ASIC perspective". *Proceedings of the 42nd annual Design Automation Conference*. ACM. 2005, pp. 275–280.

[23] Derek Chiou, Huzefa Sunjeliwala, Dam Sunwoo, John Xu, and Nikhil Patil. "Fpga-based fast, cycle-accurate, full-system simulators". *Proceedings of the second Workshop on Architecture Research using FPGA Platforms, held in conjunction with HPCA-12, Austin, TX*. 2006.

[24] Niket K Choudhary. *FabScalar: Automating the design of superscalar processors, PhD thesis*. North Carolina State University, 2012.

[25] Joel Coburn, Srivaths Ravi, and Anand Raghunathan. "Power emulation: a new paradigm for power estimation". *Proceedings of the 42nd annual Design Automation Conference*. ACM. 2005, pp. 700–705.

[26] *Coremark EEMBC Benchmark*. `https://www.eembc.org/coremark/`.

[27] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[28] Jonathan Donner. "The use of mobile phones by microentrepreneurs in Kigali, Rwanda: Changes to social and business networks". *Information Technologies & International Development* 3.2 (2006), pp–3.

[29] Brandon H Dwiel, Niket Kumar Choudhary, and Eric Rotenberg. "FPGA modeling of diverse superscalar processors". *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 188–199.

[30] Schuyler Eldridge, Amos Waterland, Margo Seltzer, Jonathan Appavoo, and Ajay Joshi. "Towards General-Purpose Neural Network Computing". *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE. 2015, pp. 99–112.

[31] Phil Emma. *Branch Prediction: Caveats and Second-Order Effects*. `https://www.jilp.org/cbp/Phil-slides.PDF`. Dec. 2004.

[32] *ESESC: A Fast Multicore Simulator*. `https://github.com/masc-ucsc/esesc/tree/aafe01b48b41d79748f299ca94cfaaaf60e7f606`.

[33] *FabScalar pre-release tools*. `http://people.engr.ncsu.edu/ericro/research/fabscalar/pre-release.htm`.

[34] Brian A Fields, Rastislav Bodík, Mark D Hill, and Chris J Newburn. "Using interaction costs for microarchitectural bottleneck analysis". *MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE. 2003, pp. 228–239.

[35] Michael J Flynn. "Some Reflections on Computer Engineering: 30 Years after the IBM System 360 Model 91". the International Symposium on Microarchitecture. Durham, NC, Dec. 1997.

[36] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. `http://www.agner.org/optimize/instruction_tables.pdf`. Technical University of Denmark, 2017.

[37] *Geekbench: Processor Benchmarks*. `https://browser.primatelabs.com/processor-benchmarks`.

[38] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. "Interval Simulation: Raising the Level of Abstraction in Architectural Simulation". *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Feb. 2010, pp. 307–318.

[39] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors". *SIGARCH Comput. Archit. News* 18.2SI (May 1990), pp. 15–26. ISSN: 0163-5964. DOI: 10.1145/325096.325102.

[40] Mohammad Ali Ghodrat, Kanishka Lahiri, and Anand Raghunathan. "Accelerating system-on-chip power analysis using hybrid power estimation". *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*. IEEE. 2007, pp. 883–886.

[41] Peter Greenhalgh. *Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7*. https://www.eetimes.com/document.asp?doc_id=1279167. Oct. 2011.

[42] Gregory F. Grohoski. "Machine organization of the IBM RISC System/6000 processor". *IBM Journal of Research and Development* 34.1 (1990), pp. 37–58.

[43] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. "Sniper: scalable and accurate parallel multi-core simulation". *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance, Embedded Architecture, and Compilation Network of Excellence (HiPEAC). 2012, pp. 91–94.

[44] ARM Holdings. "Cortex-A9: Technical Reference Manual (revision: r4p1)" (2012).

[45] Urs Hölzle. "Brawny cores still beat wimpy cores, most of the time". *IEEE Micro* 30.4 (2010).

[46] Mark Horowitz, Elad Alon, Dinesh Patil, Samuel Naffziger, Rajesh Kumar, and Kerry Bernstein. "Scaling, power, and the future of CMOS". *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*. IEEE. 2005, 7–pp.

[47] Wei-Wu Hu, Ji-Ye Zhao, Shi-Qiang Zhong, Xu Yang, Elio Guidetti, and Chris Wu. "Implementing a 1GHz four-issue out-of-order execution microprocessor in a standard cell ASIC methodology". *Journal of Computer Science and Technology* 22.1 (2007), pp. 1–14.

[48] Bunnie Huang. *Impedance Matching Expectations Between RISC-V and the Open Hardware Community*. https://riscv.org/wp-content/uploads/2017/05/Wed1100-impedancematch-huang.pdf. May 2017.

[49] Daniel A Jiménez. "Multiperspective Perceptron Predictor with TAGE". *5th JILP Workshop on Computer Architecture Competitions* (2016).

[50] Daniel A Jiménez and Calvin Lin. "Dynamic branch prediction with perceptrons". *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE. 2001, pp. 197–206.

[51] John Morris. *Chipmakers announce 7nm technology*. `http://www.zdnet.com/article/chipmakers-announce-7nm-technology/`. ZDNet.

[52] Jose Renau. *ESESC: A Fast Multicore Simulator*. `http://masc.soe.ucsc.edu/esesc/resources/tutorial13.pdf`.

[53] Karl Rupp. *40 Years of Microprocessor Trend Data*. `http://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`.

[54] Ben Keller, Martin Cochet, Brian Zimmer, Jaehwa Kwak, Alberto Puggelli, Yunsup Lee, Milovan Blagojević, Stevo Bailey, Pi-Feng Chiu, Palmer Dabbelt, et al. "A RISC-V Processor SoC With Integrated Power Management at Submicrosecond Timescales in 28 nm FD-SOI". *IEEE Journal of Solid-State Circuits* (2017).

[55] R.E. Kessler. "The Alpha 21264 Microprocessor". *IEEE Micro* 19.2 (1999), pp. 24–36. ISSN: 0272-1732. DOI: `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=755465`.

[56] 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". *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press. 2016, pp. 128–139.

[57] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanovic, and Krste Asanović. "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators". *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*. IEEE. 2014, pp. 199–202.

[58] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures". *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 469–480.

[59] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". *ACM SIGPLAN Notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[60] Heather Mackey. *Sneak peek: inside NVIDIA's Emulation Lab*. `https://blogs.nvidia.com/blog/2011/05/16/sneak-peak-inside-nvidia-emulation-lab/`. May 2011.

[61] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. "Simics: A full system simulation platform". *Computer* 35.2 (2002), pp. 50–58.

[62] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset". *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[63] Eric McCorkle. *RISC-V (Crypto) Engines Extension*. `https://ericmccorkleblog.wordpress.com/2017/02/10/risc-v-crypto-engines-extension/`. Eric McCorkle's Blog, Feb. 2017.

[64] Daniel S McFarlin, Charles Tucker, and Craig Zilles. "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" *ACM SIGPLAN Notices*. Vol. 48. 4. ACM. 2013, pp. 241–252.

[65] Scott McFarling. *Combining branch predictors*. Tech. rep. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[66] Harlan McGhan. "Niagara 2 opens the floodgates". *Microprocessor Report* 20.11 (2006), pp. 1–12.

[67] Mesa-Martinez, Francisco J, et al. "SCOORE santa cruz out-of-order RISC engine, FPGA design issues". *Workshop on Architectural Research Prototyping (WARP), held in conjunction with ISCA-33*. 2006, pp. 61–70.

[68] Pierre Michaud. "A PPM-like, tag-based branch predictor". *Journal of Instruction Level Parallelism* 7.1 (2005), pp. 1–10.

[69] 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". *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE. 2010, pp. 1–12.

[70] Christopher Mims. "The High Cost of Upholding Moore's Law". *MIT Technology Review* (Apr. 2010).

[71] MIPS Technologies, Inc. *MIPS R10000 Microprocessor Users Manual*. Mountain View, CA, 1996.

[72] Gordon E Moore. "Cramming more components onto integrated circuits". *Electronics* (Apr. 1965), pp. 114–117.

[73] Ravi Nair. "Dynamic path-based branch correlation". *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press. 1995, pp. 15–23.

[74] *OpenSPARC T2*. `http://www.oracle.com/technetwork/systems/opensparc/opensparc-t2-page-1446157.html`.

[75] Pablo Montesinos Ortego and Paul Sack. "SESC: SuperESCalar simulator". *17 th Euro micro conference on real time systems (ECRTS05)*. 2004, pp. 1–4.

[76] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. "MARSS: a full system simulator for multicore x86 CPUs". *Proceedings of the 48th Design Automation Conference*. ACM. 2011, pp. 1050–1055.

[77] David A Patterson and John L Hennessy. *Computer Architecture: a Quantitative Approach (Fifth Edition)*. Elsevier, 2012.

[78] David A Patterson and John L Hennessy. *Computer Architecture: a Quantitative Approach (Sixth Edition)*. Elsevier, 2017.

[79] Paul Alcorn. *Intel Xeon E5-2600 v4 Broadwell-EP Review*. `http://www.tomshardware.com/reviews/intel-xeon-e5-2600-v4-broadwell-ep,4514-2.html`. Tom's Hardware, Mar. 2016.

[80] Yogesh Ramadass. "Powering the Internet of Things". *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE. 2014, pp. 1–50.

[81] *RISC-V Tests*. `https://github.com/riscv/riscv-tests`. RISCV Foundation.

[82] *RISC-V Torture*. `https://github.com/ucb-bar/riscv-torture`. UCB-BAR.

[83] *Rocket Microarchitectural Implementation of RISC-V ISA*. `https://github.com/ucb-bar/rocket`. 2016.

[84] E. Rotenberg, B.H. Dwiel, E. Forbes, Zhenqian Zhang, R. Widialaksono, R. Basu Roy Chowdhury, N. Tshibangu, S. Lipa, W.R. Davis, and P.D. Franzon. "Rationale for a 3D heterogeneous multi-core processor". *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. Oct. 2013, pp. 154–168. DOI: `10.1109/ICCD.2013.6657038`.

[85] Daniel Sanchez and Christos Kozyrakis. "ZSim: fast and accurate microarchitectural simulation of thousand-core systems". *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 475–486.

[86] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. "Intel Nehalem Processor Core Made FPGA Synthesizable". *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '10. Monterey, California, USA: ACM, 2010, pp. 3–12. ISBN: 978-1-60558-911-4. DOI: `10.1145/1723112.1723116`.

[87] Frank Schirrmeister. *Productivity, Predictability, and Use-Model Versatility: The Three Key "Care-Abouts of Choosing Hardware-Assisted Verication*. Tech. rep. Cadence Design Systems, 2013.

[88] Carlo H Séquin and David A Patterson. *Design and Implementation of RISC I*. Tech. rep. Computer Science Division, University of California, 1982.

[89] André Seznec. "A 256 kbits l-tage branch predictor". *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* (2007).

[90] André Seznec. "A new case for the TAGE branch predictor". *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 117–127.

[91] André Seznec. "Tage-sc-l branch predictors". *JILP-Championship Branch Prediction*. 2014.

[92] André Seznec. "Tage-sc-l branch predictors again". *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. 2016.

[93] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. "Design tradeoffs for the Alpha EV8 conditional branch predictor". *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE. 2002, pp. 295–306.

[94] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. *Multiple-block ahead branch predictors*. Vol. 31. 9. ACM, 1996.

[95] Anand Lal Shimpi. *NVIDIA's Fermi: Architected for Tesla, 3 Billion Transistors in 2010*. `http://www.anandtech.com/show/2849`. Sept. 2009.

[96] Clay Shirky. "The political power of social media: Technology, the public sphere, and political change". *Foreign affairs* (2011), pp. 28–41.

[97] James E Smith. "A study of branch prediction strategies". *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press. 1981, pp. 135–148.

[98] *SPEC CPU2006: dynamic instruction counts*. `http://boegel.kejo.be/ELIS/spec_cpu2006/spec_cpu2006_dyn_instr_counts.html`.

[99] Jayanth Srinivasan. *An overview of static power dissipation*. Tech. rep.

[100] Chen Sun, Mark T Wade, Yunsup Lee, Jason S Orcutt, Luca Alloatti, Michael S Georgas, Andrew S Waterman, Jeffrey M Shainline, Rimas R Avizienis, Sen Lin, et al. "Single-chip microprocessor that communicates directly using light". *Nature* 528.7583 (2015), pp. 534–538.

[101] Dam Sunwoo, Gene Y Wu, Nikhil A Patil, and Derek Chiou. "PrEsto: An FPGA-accelerated power estimation methodology for complex systems". *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE. 2010, pp. 310–317.

[102] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. "RAMP gold: an FPGA-based architecture simulator for multiprocessors". *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE. 2010, pp. 463–468.

[103] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. "A case for FAME: FPGA architecture model execution". *ACM SIGARCH Computer Architecture News* 38.3 (2010), pp. 290–301.

[104] Michael B Taylor. "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse". *Design Automation Conference (DAC), 2012 49th ACM/-EDAC/IEEE*. IEEE. 2012, pp. 1131–1136.

[105] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, et al. "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams". *ACM SIGARCH Computer Architecture News* 32.2 (2004), p. 2.

[106] *The Hwacha Project*. `http://hwacha.org`. 2015.

[107] *The RISC-V Instruction Set Architecture*. `http://riscv.org/`.

[108] James E Thornton. "Parallel operation in the Control Data 6600". *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*. ACM. 1964, pp. 33–40.

[109] Matthew Travers. "CPU Power Consumption Experiments and Results Analysis of Intel i7-4820K" (2015).

[110] Jessica H Tseng and Krste Asanović. "Energy-efficient register access". *Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on*. IEEE. 2000, pp. 377–382.

[111] Jessica Hui-Chun Tseng. *Banked Microarchitectures for Complexity-Effective Superscalar Microprocessors, PhD thesis*. Massachusetts Institute of Technology, 2006.

[112] Stuart G Tucker. "Emulation of large systems". *Communications of the ACM* 8.12 (1965), pp. 753–761.

[113] Jim Turley. "Cortex-A15 "Eagle" flies the coop". *Microprocessor Report* 24.11 (2010), pp. 1–11.

[114] *UCB-BAR: Berkeley Hardfloat*. `https://github.com/ucb-bar/berkeley-hardfloat`.

[115] UN News Centre. *Deputy UN chief calls for urgent action to tackle global sanitation crisis*. `http://www.un.org/apps/news/story.asp?NewsID=44452`.

[116] UNdata. *mobile-cellular telephone subscriptions*. `http://data.un.org/Data.aspx?d=ITU&f=ind1Code%3aI271`.

[117] *Verilator*. `https://www.veripool.org/wiki/verilator`.

[118] Narayanan Vijaykrishnan, Mahmut Kandemir, Mary Jane Irwin, Hyun Suk Kim, and Wu Ye. "Energy-driven integrated hardware-software optimizations using SimplePower". *ACM SIGARCH Computer Architecture News* 28.2 (2000), pp. 95–106.

[119] Nicholas J Wang, Justin Quek, Todd M Rafacz, and Sanjay J Patel. "Characterizing the effects of transient faults on a high-performance processor pipeline". *Dependable Systems and Networks, 2004 International Conference on*. IEEE. 2004, pp. 61–70.

[120] James Warnock, Brian Curran, John Badar, Gregory Fredeman, Donald Plass, Yuen Chan, Sean Carey, Gerard Salem, Friedrich Schroeder, Frank Malgioglio, et al. "4.1 22nm next-generation IBM System Z microprocessor". *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*. IEEE. 2015, pp. 1–3.

[121] Thomas F Wenisch, Roland E Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C Hoe. "SimFlex: statistical sampling of computer system simulation". *IEEE Micro* 26.4 (2006), pp. 18–31.

[122] Steven JE Wilton and Norman P Jouppi. "CACTI: An enhanced cache access and cycle time model". *IEEE Journal of Solid-State Circuits* 31.5 (1996), pp. 677–688.

[123] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. "Quantifying sources of error in McPAT and potential impacts on architectural studies". *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE. 2015, pp. 577–589.

[124] K.C. Yeager. "The MIPS R10000 Superscalar Microprocessor". *IEEE Micro* 16.2 (1996), pp. 28–41. ISSN: 0272-1732. DOI: `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=491460`.

[125] Charles Zhang. "Mars: A 64-Core ARMv8 Processor". *Hot Chips*. Phytium. 2015.

[126] Yanqi Zhou and David Wentzlaff. "The sharing architecture: sub-core configurability for IaaS clouds". *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM. 2014, pp. 559–574.

[127] Brian Zimmer, Pi-Feng Chiu, Borivoje Nikolić, and Krste Asanović. "Reprogrammable Redundancy for SRAM-Based Cache Vmin Reduction in a 28-nm RISC-V Processor". *IEEE Journal of Solid-State Circuits* (2017).