Physically Aware Design of Generated Systems-on-Chip

by

John Charles Wright

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolić, Chair
Professor Elad Alon
Professor Robert Leachman

Summer 2021

Physically Aware Design of Generated Systems-on-Chip

Abstract

Physically Aware Design of Generated Systems-on-Chip

by

John Charles Wright

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Generator-based integrated-circuit design flows are crucial for meeting the aggressive system-on-chip development timelines demanded by rapidly changing modern workloads. Generators allow chip designers to develop complex solutions to classes of problems rather than individual instances, allowing significant design changes late in the development cycle and enabling incremental improvements to existing solutions. The philosophies of welcoming changing requirements and steady, incremental development have been embraced by the software development community for some time, but have only recently been incorporated into hardware development. While this adoption of proven software development philosophies has decreased the turnaround time of systems-on-chip, productivity has been limited by hard-to-automate tasks like physical design. Each generated design instance requires a new human-generated floorplan or other changes to the physical design flow, limiting the throughput of design space exploration by the available engineering resources. Automation of physical design is therefore critical for state-of-the-art generator-based system-on-chip design.

This work describes a series of generator-based integrated circuits manufactured in 28nm FD-SOI and 16nm FinFET, outlines the physical design challenges encountered in their development, and presents a physical design methodology purpose-built to solve these challenges. The integrated circuits presented include an 8192-point digital spectrometer in 28nm FD-SOI, a dual-core RISC-V vector processor with on-chip fine-grain power management in 28nm FD-SOI, a dual-lane RISC-V vector processor with a dedicated on-chip power management core in 28nm FD-SOI, an eight-core RISC-V vector machine in 16nm FinFET, and a 21-core RISC-V vector machine with a systolic array accelerator in 16nm FinFET. The eight-core chip achieves a state-of-the-art energy efficiency of 209.5 GFLOPS/W on a half-precision matrix multiplication (GEMM) kernel.

The physical design methodology presented uses a framework, Hammer, to provide reusable physical design deliverables by decoupling the design-specific, tool-specific, and technology-specific aspects of back-end design along with a novel floorplan generation framework for

Chisel designs. This physical design methodology has been incorporated into the Chipyard framework, an open-source RISC-V system-on-chip development platform leveraging the Chisel hardware construction language. The floorplan generation framework allows Chisel programs, which generate RTL, to specify composable floorplans without modifying the original source code. The flow solves common challenges associated with floorplanning generated RTL, such as SRAM mapping and placement, demonstrating the efficacy of floorplan generation in reducing the overhead and cycle times of generator-based design.

To my wife, Xuân, and our unborn son.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgments

I am tremendously grateful for the help and support I received throughout my doctoral studies. Berkeley is uniquely collaborative, and I am fortunate to have worked with numerous brilliant minds during my time here. While I am appreciative of everyone I met throughout this adventure, the following colleagues, mentors, faculty, staff, and friends deserve special recognition. I apologize in advance to any deserving individuals that I may have missed.

First and foremost, I am grateful to have had Bora Nikolić as an advisor. Bora, you are uniquely skilled in identifying problems that are simultaneously interesting to industry and academia. I continue to be impressed by your breadth and depth of knowledge, your ability to keep up with the state of the art, and your ability to drive projects of massive scope to success. There is simply no way I could have had the opportunity to tape out so many times, in such advanced process nodes, or such large chips without your leadership. I believe your continuing push for agile hardware development is producing a generation of ASIC designers who think so differently that we will feel the impact industry-wide. I will miss your ComIC group meeting stories and will have to find some new way of keeping up with the latest industry gossip.

I am thankful to have had additional guidance from Elad Alon and Krste Asanović. Elad, your support throughout the Eagle project was incredibly helpful, especially when the hours grew long and deadlines started passing. I appreciate your willingness to jump on Slack calls at 4 A.M. to hack on RTL alongside the grad students to keep the tapeout moving. I also enjoyed working with and learning from you and your students on the serial link generator project. Krste, thank you for your support throughout all the tapeouts and project meetings and for making me feel welcome to participate in projects with the Berkeley Architecture Research group. I eagerly look forward to seeing what's next for RISC-V. I would also like to thank Vladimir Stojanović for serving on my qualifying exam committee, reading my M.S., and otherwise being available for discussion. Likewise, I thank Rob Leachman for serving on my qualifying exam and dissertation committees and Jonathan Bachrach and Sophia Shao for their help along the way.

I am indebted to a number of people in the broader engineering community for their guidance as well. Todd Hastings and Bruce Walcott (University of Kentucky), thank you for getting me started in research and sending me down this path. Scott Savage and Joseph Elias (Infineon), thank you for providing recommendations and encouragement. It was a tough decision to leave a full-time job to return to school, and I appreciate your candid advice and assistance to do what you believed to be in my best interest. Mohamed Abu-Rahma and Jared Zerbe (Apple), thank you for your mentorship and the thought-provoking technical discussions. The opportunity to work on cutting-edge projects at Apple provided meaningful insight and context to the work I did at Berkeley. Mark Rowland and Steve Burns (Intel), your feedback at the many ASPIRE and ADEPT retreats was useful and entertaining. It's not always clear that we are moving in the right direction, and confirmation (bonus points) from our industrial sponsors is always welcome. Rajeev Jain (Qualcomm), your affirmation of the usefulness of many components of this dissertation was rewarding as well. I appreciate the deep technical discussions I had with you and others at Qualcomm in the denouement of my graduate career. Masood Qazi (Qualcomm), thank you for the advice you gave along the way. I'm happy to have overlapped with you at Cypress and Apple and hope we'll one day overlap again.

Much of the work presented in this dissertation required custom fabrication and assembly of chips, packages, circuit boards, et cetera. There are many who helped with this, but I owe a few individuals special mentions; thank you. Mo Ohady (Digicom Electronics) was available off-hours for special assembly requests to meet paper deadlines and always enthusiastic to work with Berkeley. Gary Thorne (MOSIS) helped with our GDS submissions, and I would especially like to thank him for enduring so many slips and being available for the final EagleX submission on the 4th of July holiday. Linton Salmon (DARPA) led the CRAFT program, which made a large portion of this work possible. Andreia Cathelin (STMicroelectronics) championed many of the earlier tapeouts in my graduate career (Splash2, Hurricane1, Hurricane2). Hien Ly, Quyen Chu, and Sundar Sethuraman (Jabil Blue Sky) helped by experimenting with aggressive die attach techniques and gave us a lab tour. Didier Campos and Vince Mangion (STMicroelectronics) designed the Hurricane2 package. Pierre Brunet (EuroCan) designed the EagleX package when many others would not. Darin Heckendorn (Cadence) provided invaluable support of the tool flow for Eagle and EagleX.

Within Berkeley, we are lucky to have talented research staff to help ensure the success of these projects. Brian Richards is a vault of tapeout knowledge and spent many hours helping with every project I worked on. Thank you for all you do to keep the BWRC tapeout machine running. Anita Flynn is a PCB wizard, human confusion detector, and bug catcher. Thank you for all your hard work on the Hurricane2, Eagle, and EagleX boards; they are works of art. James Dunn, your curiosity and excitement is enviable, and I hope you continue to have such passion. Thank you for helping with sysadmin, lab, and PCB tasks over the years. Shirley Salanio does so much behind-the-scenes work to help every EECS grad student stay on track with program requirements. Thank you for all your help over the years, and especially for helping with the flurry of questions about Summer filing procedures. Candy Corpus and Yessica Bravo, thanks for organizing social events and generally making BWRC a happy place. Greg Pearson and Jeff Anderson-Lee, thanks for keeping the compute servers up and running despite my best efforts to fill up the file system. Chick Markley, thanks for all the interesting discussions and scala whispering. I also thank Kostadin Ilov, Ria Briggs, Tami Chouteau, Fred Burghardt, Mikaela Cavizo-Briggs, Ken Lutz, Melissa Trevizo, Olivia Nolan, Amber Sanchez, and Erin Hancock.

A large portion of this work was done in collaboration with Colin Schmidt, who has become a great friend and a colleague I hope to continue working with in the future. Somehow after all of the all-night Slack calls, frustrating bring-up puzzles, and tapeout delays, you still kept a positive attitude. I'm indebted to you for all the help you provided, all your code reviews, and support throughout the whole process.

I also thank the following individuals for their technical contributions. Dan Werthimer and Robert Jarnot contributed significantly to the Splash2 architecture and guided the project. Zhongkai Wang drove the SerDes generator development for the Eagle tapeout with the help of Eric Chang, Paul Kwon, and Woorham Bae. Howie Mao wrote a large number of RTL widgets we used for every tapeout. Adam Izraelevitz built Chisel Aspects, which is an integral component of the floorplan generator. Albert Ou wrote RTL and software and helped with bring-up for the Hurricane and Eagle tapeouts and was also willing to answer many of my silly RISC-V and Linux questions. Edward Wang led the Hammer project and designed and built much of its infrastructure, which made the Eagle, EagleX, and floorplanning work in this dissertation possible. Vighnesh Iyer helped with many different FPGA designs and automated the build flow. Jingyi Xu, Ryan Lund, Anson Tsai, and Gary Choi were wonderful undergraduate and master's researchers who helped with various aspects of these projects.

I'm happy to have had the support of many other friends and colleagues at BWRC throughout this journey as well. Antonio Puglielli, thank you for being a dependable backpacking adventure-buddy and jib-trimmer. Krishna Settaluri, thank you for being a good motivator and for all the lighthearted jokes. Luke Calderin, you are a fantastic co-skipper of The Floating Node, even though we never did sail to a retreat, and for that I salute you. Nathan Narevsky, thanks for always showing up early to help set up the party and for enduring Zoom thesis-writing sessions with me. Ozzy LaCaille, thank you for the coffee walks, the complaining sessions, and the Tide Pod. Sameet Ramakrishnan, thanks for all the brainstorming sessions, entertaining sports bets, and, of course, getting me a job. Pavan Bhargava, thanks for always bringing a delicious dish and for joining in our adventures. Stevo Bailey, thanks for dragging me to Core Blast, for showing me the ropes in the early days, and for your upbeat attitude. Ben Keller, thanks for your leadership on the Hurricane projects, for your activism, and for taking all my extra tomato plants. Alon Amid, in addition to all your help with paper writing and Chipyard, thanks for introducing us to Bamba and always bringing an interesting story to Thanksgiving. Paul Rigge, thanks for always having an interesting perspective and for explaining scala nuances in great detail. Keertana Settaluri, thanks for being my over-the-divider buddy. Daniel Grubb and Harrison Liew, I appreciate your senses of humor; thank you for joining us in the trenches of the 2019 tapeouts and for carrying the baton to keep many of these projects alive. Rachel Hochman, thank you for throwing fancy parties and always being supportive and positive. Nandish Mehta, thank you for providing many laughs and, of course, your dal makhani. Mira Videnović-Mišić, thank you for your genuine kindness and always bringing delicious treats to share. Ajith Amerasekera, thank you for always being available for a quick chat at Triple Rock.

There are many other BWRC students and visitors who impacted my time here, and I'd like to thank a few of them as well: Brian Zimmer, Jaehwa Kwak, Amy Whitcombe, Milos Jorgovanović, Pi-Feng Chiu, Martin Cochet, Jarno Salomaa, Guillaume Bonnechère, Angie Wang, Matthew Anderson, Lorenzo Iotti, Nick Sutardja, Katerina Papadopoulou, Marko Kosunen, Sean Huang, Sam Steffl, Zhenghan Lin, Yue Dai, Zhaokai Liu, Vladimir Milovanović, Emily Naviasky, Christos Adamopoulos, Sidney Buchbinder, Andy Zhou, and Chris Yarp. Thanks for your help, and thanks for keeping BWRC lively. I'd also like to acknowledge some additional members of the ASPIRE/ADEPT labs: Abe Gonzalez, Jerry

Zhao, Sagar Karandikar, Albert Magyar, David Biancolin, Nathan Pemberton, Hasan Genc, Ameer Haj-Ali, Eric Love, Kevin Laeufer, Yunsup Lee, Andrew Waterman, Palmer Dabbelt, and Jack Koenig. Thanks for making my up-the-hill time enjoyable.

Outside of the Berkeley community, I have been fortunate to have the support of many loving friends and family, most of whom are not explicitly named here but are appreciated nonetheless. I would like to thank Alex Heilman specifically for his encouragement to pursue a Ph.D. and his support throughout. My parents, Bonnie and Randy, instilled a respect and admiration of higher education in me from a young age. I appreciate their unrelenting encouragement throughout this long endeavor. I also think my brother, Clay, for coming to visit and keeping me upbeat.

Finally, I owe all of this to my loving wife and life co-captain, Xuân. She not only endured all of the all-nighters, the months-long tapeout pushes, retreats, quals, etc., but she also provided much-needed graduate school advice, helped me edit papers, and critiqued my figures along the way. I appreciate your sacrifices and understanding that helped get me through this, and look forward to our next big step coming very soon. I love you and thank you.

Funding

I thank the sponsors of this work for their financial support and for providing multiple opportunities to tape out on expensive process nodes. The Splash2 project was supported by NASA's Earth Science Technology Office Instrument Incubator Program as part of the Compact, Adaptable Microwave Limb Sounder project, grant number NNX12AK39G. The Hurricane1 and Hurricane2 projects were supported in part by Defense Advanced Research Projects Agency (DARPA) Power Efficiency for Embedded Computing Technologies (PERFECT) award HR0011-12-2-0016 and in part by the Intel Science and Technology Centers (ISTC) on Agile Design. Fabrication of Splash2, Hurricane1, and Hurricane2 was donated by STMicroelectronics. The Eagle, EagleX, and Hammer projects were supported by DARPA Circuit Realization at Faster Timescales (CRAFT) grant HR0011-16-C0052 and the Intel ISTC on Agile Design. The Chipyard project was funded by NSF grant number 2016662. I also thank the sponsors of the Berkeley Wireless Research Center, ASPIRE, and ADEPT labs for additional funding that supported this work.

Contributions to this work

RocketChip is an open-source IP initially developed at UC Berkeley and now maintained by SiFive. Chisel and FIRRTL are open-source projects developed and maintained by numerous people at Berkeley and elsewhere. The Splash2 ASIC development was led by Stevo Bailey. Vladimir Milovanović contributed the PLL design. Nandish Mehta contributed the serial link receiver design for Splash2, Hurricane1, and Hurricane2. The Hurricane1 and Hurricane2 ASIC development was led by Ben Keller. Cadence Design Systems donated the DDR PHY for Hurricane2, and the DC-DC converters were reused from prior projects. The Eagle

and EagleX projects were co-led with Colin Schmidt. Sean Huang built the PLL based on previous work by Vladimir Milovanović. Zhongkai Wang, Eric Chang, and Woorham Bae built the serial links for Eagle and EagleX. Colin Schmidt and Albert Ou developed the Hwacha accelerator for Hurricane1, Hurricane2, Eagle, and EagleX. SiFive donated the L2 and L3 cache generator RTL for Eagle and EagleX. Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, and Alon Amid developed the Gemmini accelerator for EagleX. The Hammer project was led by Edward Wang.

Chapter 1

Introduction

1.1 Motivation

Throughout the second half of the twentieth century, integrated circuit (IC) designers enjoyed a consistent improvement in process technology as famously predicted by Gordon Moore [1]. This period saw everything from the first integrated circuits with only a handful of transistors visible by the naked eye to quarter-micron microprocessors running at sub-nanosecond clock periods [2]. While Moore's law continued into the early twenty-first century, another once-dependable law authored by Robert Dennard began to break down. In his paper [3], Dennard noted that the increases to power caused by increasing transistor density and frequency were offset by decreasing supply voltage. In the mid-2000s, Dennard scaling began to slow due to the increasing effect of leakage power at short channel lengths, a phenomenon also known as the "Power wall" [4].

Reaching the power wall forced IC designers to improve performance in new and creative ways—primarily via the introduction of multicore processor chips [5]. As state-of-the-art transistor feature sizes approach fundamental limits, it is reasonable to assume the end of Moore's Law is also imminent and that similar creativity will be required to continue advancing processing capability. This, along with the surge in compute-intensive workloads like deep neural networks, has led to increased use of specialized compute accelerators on modern systems-on-chip (SoCs) [6–9]. With the changes to these workloads outpacing typical IC design cycles, it is becoming increasingly necessary to make improvements to IC design methodology to shorten these design cycles [10–12].

Generators, software programs capable of producing instances from a larger class of possible configurations, have been shown repeatedly to improve IC design productivity [13–15]. When building digital components, the primary focus of generators thus far has been to produce logical circuit descriptions (register-transfer level, or RTL) and any necessary verification and integration collateral. When generating only a small component of a larger system, this is often sufficient, as the generated component is not likely to require a custom floorplan. However, when generating more complex components, components with large numbers of memory macrocells, or full SoC designs, the overall cycle time improvement offered by the RTL generator is limited by the manual back-end implementation cycle, which is frequently dominated by the creation of an instance-specific floorplan. It is therefore desirable that generators should not only produce logical descriptions of configured instances but also produce a floorplan for each generated instance in order to maximize the productivity gains afforded by generator-based design.

This dissertation presents multiple systems-on-chip manufactured in advanced process nodes build using generator-based design flows. The lessons learned from these tapeouts are motivate a physical design methodology, which is also presented. This physical design methodology has been incorporated into the open-source Chipyard [16] framework and includes both a physical design generator framework, Hammer [17], and a novel floorplanning framework for Chisel designs.

1.2 Background

1.2.1 Fundamentals of VLSI design

Very-large-scale integration (VLSI) is the process of manufacturing integrated circuits (ICs) with large numbers of transistors and has existed in different forms since the 1970s. In a traditional design flow, the VLSI design process begins with the creation a high-level architectural description of the entire system, including both on-chip and off-chip elements. The architectural description includes every aspect of the design required for software programmers to use the system without specific implementation details. Next, one or more microarchitectures are created to implement the specified architecture. These microarchitectures include the circuit-level implementation details, down to the signal level, needed to create a functioning design. An implementation team uses this information to build a logical schematic, a detailed description of devices[1] and their connections, and a layout, a representation of physical geometries that implements them.

The semiconductor integrated circuit manufacturing process consists of a series of steps which manufacture the devices themselves, called front-end processing, and the metal wires that connect them, called back-end processing. Each process step may use one or more photomasks to restrict the affected area to the unique geometries for a specific design. The set of geometries for a single photomask and processing step combination is often termed a layer, or, less commonly, level. Thus, the layout for an integrated circuit design will consist of geometries on multiple layers.

The layers in a layout are specific to a process technology, often shortened to process or technology, meaning that the geometries on them are generally are not portable to other process technologies[2]. A process technology is a series of repeatable recipes used to manufac-

[1]In digital designs, a device often refers to a transistor, but integrated circuits usually include other classes of devices, including resistors, inductors, capacitors, fuses, and diodes.

[2]This has a number of implications on the reusability of design work, which are discussed in Chapter 4.

ture integrated circuits in a semiconductor fabrication plant ("Fab"). Many different circuit designs will use the same process technology with different, unique photomasks to achieve different functionality.

In the early years of semiconductor manufacturing, the integrated circuit design firm would also own the fab. While this afforded circuit design groups the luxury of highly customizable manufacturing processes, the economy of scale eventually drove a majority of the industry to use what is known as the "Fabless" model, where integrated circuits are designed and manufactured by separate entities through contract manufacturing. The contract manufacturing companies offer a menu of process technologies, amortizing the cost of research, development, and capital expenditure over a large volume of products from multiple customers.

Modern VLSI design utilizes a wide range of computer-aided design (CAD) tools to automate labor-intensive tasks that were previously performed manually. A modern tool flow for digital design primarily focuses on three tasks: synthesis, placement, and routing. A synthesis tool consumes a logical description of a circuit, typically Verilog, SystemVerilog, or VHDL, and produces a gate-level netlist, which is a textual representation of logic gate instances and their connections, called nets. These logic gates come from a process-specific library provided by the fab, developed internally, or acquired from a third party. These libraries contain combinational and sequential logic elements, standard cells, built from transistors and are designed so that the cells fit into a grid and can be tiled easily by an automated tool. The placement tool, or placer, will consume this netlist along with cell layout information and use heuristics to determine the placement of each logic cell within the specified die area. The placer will also replace cells with larger[3], logically equivalent cells to meet circuit timing goals specified by the user at the cost of additional power and area. The router is then responsible for creating wires to connect the placed cells using the available metal routing layers in the process technology. Placement and routing are often implemented as steps within a single CAD tool, a place-and-route tool, which gives it the ability to perform multiple iterations of incremental placement and routing to achieve area and timing goals, the latter of which is colloquially termed "Closing timing." Table 1.1 lists common CAD tool types in a typical flow.

Floorplanning also includes creation of routing blockages and routing guides, which help the tool avoid areas that create problematic routing congestion or design rule violations. Floorplans can similarly include placement blockages to prevent the tool from placing standard cells in unwanted regions, often to prevent spacing-related design rule violations.

The synthesis and place-and-route tools require additional information for macrocells included in the design. Macrocells, or macros, are circuit components that do not conform to the standard cell tiling scheme and are usually much larger than any single standard cell. Until recently, placement tools have required the user to manually determine placements for macrocells, creating what is known as a floorplan. There is a research trend moving towards

[3]Larger in this context can either mean physically larger transistors in the drive stage or transistors with a lower threshold voltage, or both.

Table 1.1: Examples of CAD tool applications.

Tool	**Purpose**	**Input**	**Output**
Synthesis	Convert RTL to gates	RTL	Gate-level netlist
Place-and-route (P&R)	Create a physical layout of gates	Gate-level netlist	Layout, Gate-level netlist
Design rule checker (DRC)	Check manufacturability of layout	Layout	Report
Layout versus schematic (LVS)	Check that layout matches schematic	Layout, Gate-level netlist	Report
RTL Verilog simulation	Simulate a design	RTL, stimulus	Waveforms
Gate-level Verilog simulation	Simulate a design	GL netlist, stimulus	Waveforms
Power analysis	Analyze dynamic and leakage power	GL netlist, waveforms	Report
SPICE simulation	Analyze analog and mixed signal circuits	SPICE netlist, stimulus	Waveforms
Characterization	Create timing models for library cells	Waveforms	.lib files
Static timing analysis (STA)	Analyze digital circuit timing	Gate-level netlist, .lib files	Reports

applying machine-learning techniques to automate this process, with reinforcement-learning techniques able to beat humans in some situations [18], but human experts are currently still competitive. A system-on-chip will include many different types of macrocells, some of which perform logical functions in the circuit and others that are necessary for electrical functionality or manufacturability. The most common macrocell with logical functionality in a digital design is a synchronous random-access memory (SRAM), which is an area- and energy-efficient circuit for storing state within a design.

After a place-and-route tool has created a layout, designers begin what is known as “Signoff,” the process of confirming that the layout is manufacturable and meets specifications.

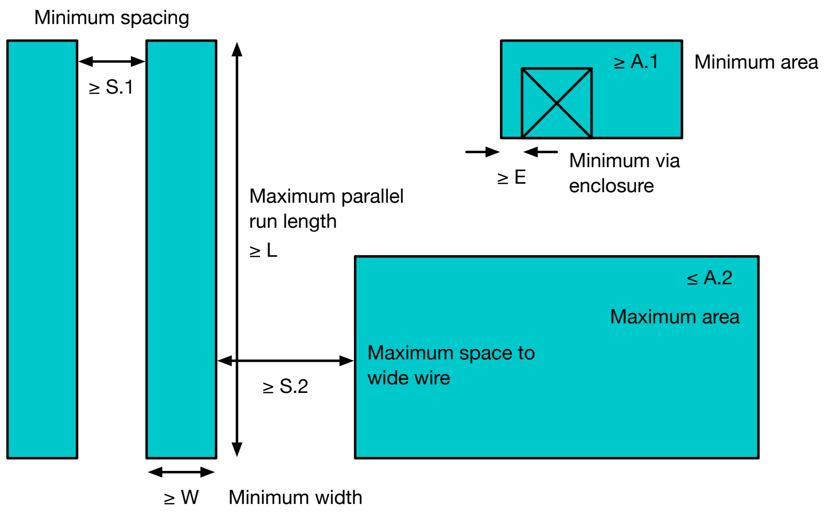


Figure 1.1: An example of metal layer design rules.

Design rule checking (DRC) checks every shape in the layout against a set of procedural rules, sometimes called a deck, to verify that the design is manufacturable. Simpler rules, like checking minimum spacing between two polygons on the same layer or checking that a single polygon meets a minimum area constraint, tend to be derived from the capabilities of the lithography process. More complex rules check for hard-to-simulate electrical issues like latch-up, an effect where parasitic bipolar transistors can cause positive feedback that destroys the chip. An example of common DRC rules for metal layers is shown in Figure 1.1, although modern process nodes have many more complex conditions than can be easily shown in a single figure. DRC is very parallelizable, but even so can take from hours to days for a full chip due to the vast number of polygons in a full chip layout. For digital flows, DRC rules tend to be related to routing, but sometimes also involve standard cell or macro placement. Placement issues are always solved with tool script changes, but routing issues are sometimes fixed by hand by correcting the tool-generated paths with custom shapes. However, most tool-generated routes for properly configured designs will have no violations.

Layout versus schematic (LVS) checks that the final layout matches the schematic, which captures the intended functionality for a chip. This tool extracts a comparison schematic from the layout and uses an algorithm to correlate the devices and nets between the two schematics. Analog components on the chip will use a SPICE netlist for the schematic, while digital components generally use Verilog to specify the gate-level netlist along with the SPICE netlists for the standard cells instantiated within. Any unintended changes, shorts, or other problems will be flagged in the tool report. When given a good floorplan and proper constraints, the place-and-route tool will produce a layout that matches the schematic, however routing congestion will lead to opens or shorts on nets that the tool cannot route properly. Custom routing, power nets, or nets inside of black boxes can also create LVS problems that the place-and-route tool is unable to fix. For these reasons, it is imperative to audit the place-and-route tool output with LVS to ensure the design will function as intended.

Static timing analysis (STA) calculates the timing paths in the design by summing the incremental cell and wire delays using pre-characterized delay models. These timing paths start from clocks or constrained inputs and end at sequential data pins or constrained outputs. Any paths that violate setup or hold constraints[4] are reported so that the designer can fix them with RTL or physical design changes. For production designs, STA is run at multiple process, voltage, and temperature (PVT) corners to ensure the design has sufficient margin to tolerate variations in manufacturing and a range of environmental conditions.

STA is also used to generate timing annotations, which are applied to gate-level Verilog simulations to simulate the implemented design while modeling real delays. This is useful for verifying that the gate level design still functions as the RTL intended and for generating accurate waveforms for power simulation. Improper RTL structures or custom changes to the netlist within the CAD tools can result in gate-level netlists which do not function identically to the RTL. Logical equivalency checking (LEC) is another tool that uses formal methods[5] to prove equivalence between gate-level netlists and RTL. There are many other signoff flows not discussed here; the specific concerns of the designers and goals for the design will determine what additional signoff is required.

1.2.2 Impact of technology scaling on system architecture

Once the 28nm process node saw widespread adoption and volume production, per-transistor costs in new nodes no longer continued to steadily and reliably decline [19, 20], shown in Figure 1.2. This has the effect of decreasing margins for comparable designs, increasing the volume required to amortize the non-recurring engineering (NRE) and capital expense required to fabricate (tape out) a new design, exacerbated by the increasing number of masks required for multiple patterning[6] [21]. Designers of new low-volume, special-purpose systems-on-chip must decide between staying at a cost-efficient process node like 28nm or by increasing price to recoup margins, which may lose customers to cost-competitive, higher-volume, general-purpose chips.

To bridge the gap between fixed-function ASICs and general-purpose processors, a growing trend is the use of highly specialized accelerators [22–24], which are compute engines tuned to a specific workload or family of workloads that can be added to a conventional system-on-chip platform to complement the general-purpose central processing units (CPUs). These accelerators vary in programmability, with highly programmable accelerators offering

[4]There are other constraint types, including recovery, removal, and minimum pulse width, which are not discussed here.

[5]Not to be confused with formal verification.

[6]Multiple patterning may be obviated by extreme ultraviolet (EUV) lithography, but this comes with its own challenges and costs.

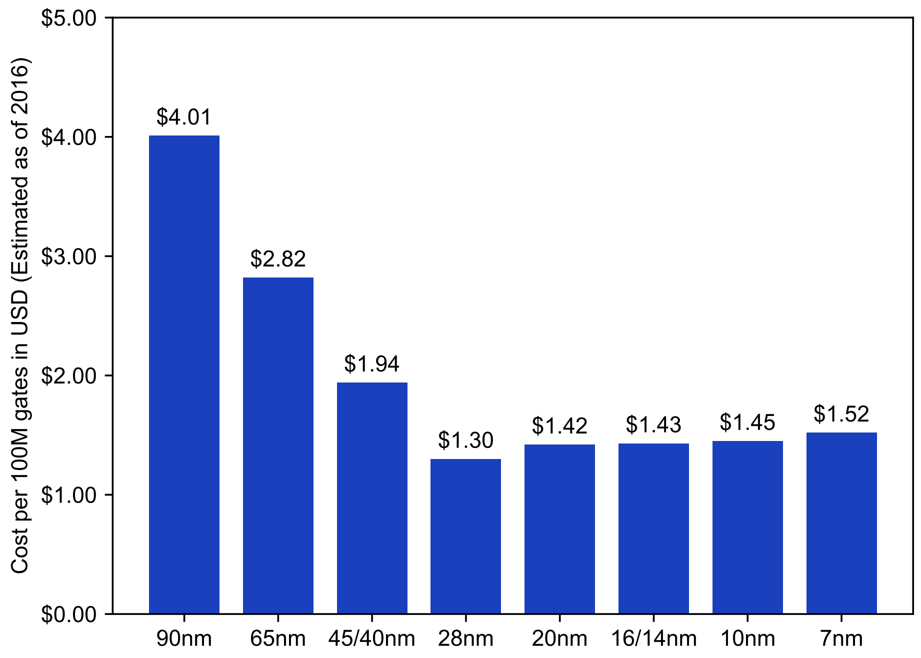


Figure 1.2: Cost per 100 million gates across process technologies as of 2016 [20].

more flexibility but lower cost and power savings than their less-flexible counterparts. While accelerators have uses in all process nodes, they are increasingly being used to improve performance in older nodes while avoiding the increased tapeout costs associated with cutting-edge nodes. This is especially useful for medium- and low-volume parts, which have trouble recouping these costs.

The CPU-with-accelerator approach is an important data point on a spectrum of solutions that provide chip designers options for trading flexibility for cost, energy-efficiency, or both for a given application. Figure 1.3 illustrates a set of design paradigms that span the range of flexibility options. General-purpose CPUs are the least energy-efficient as they rely on pure software implementations of workloads and are unable to exploit temporal locality as efficiently as specialized accelerators, requiring many data to be transferred to and from DRAM multiple times. Systems targeting aggressive workloads will also require multiple or large CPUs to meet the equivalent performance of more specialized solutions, increasing their cost. Graphics processing units (GPUs) can be more energy-efficient than CPUs on tasks with high levels of data parallelism, but this caveat inherently renders them less flexible than CPUs. Popular modern workloads like deep neural networks and blockchain technology are examples of these tasks with high levels of data parallelism.

Like GPUs, field-programmable gate arrays (FPGAs) can provide advantages over CPUs for specific workloads. Unlike GPUs and CPUs, FPGAs are not programmed with software, rather relying on the user to provide a hardware description to create custom connections on a pre-fabricated logic array. The maximum clock frequency of an FPGA is usually lower

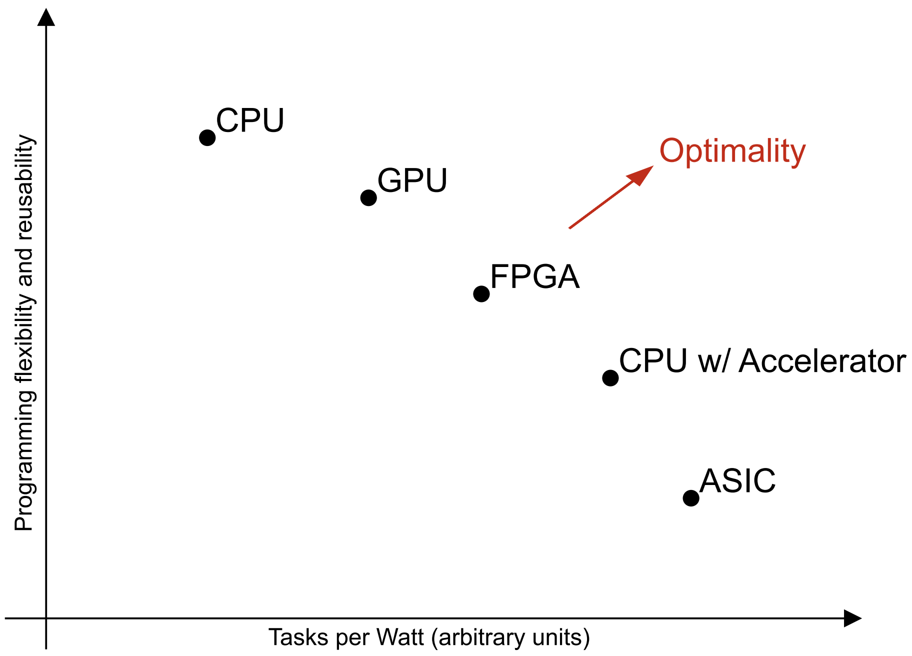


Figure 1.3: Pareto-optimal frontier of compute acceleration.

than that of a CPU or GPU—hundreds of Megahertz versus Gigahertz—but FPGAs can be configured with custom data paths that perform computation in a single cycle equivalent to many CPU instructions. These FPGA designs can be used in products, but for a given number of logic gates, FPGAs are much larger and more expensive than application-specific integrated circuits (ASICs) due to the configuration overhead, so products which use FPGAs tend to be very low volume. While FPGAs are technically very flexible by virtue of being highly configurable, the difficulty of programming them and the relatively low number of people with the required skill set makes them a more specialized approach than either CPU or GPU software programming. Purpose-built ASICs offer the highest level of customization, performance, and energy-efficiency by offering the same custom data paths available to FPGAs but with much higher clock frequencies and lower power. ASICs, however, cannot be reconfigured in the field like FPGAs, making them the least flexible of all design paradigms. The CPU-with-accelerators scheme combines aspects of ASICs with general-purpose CPUs, resulting in a design that is slightly more flexible than a pure ASIC, but with additional software programmability. By effectively utilizing this gamut of design styles, chip designers can design in process nodes behind the leading edge to produce high-performance parts that meet specific needs while maintaining cost efficiency.

1.2.3 Generator-based design

Generator-based design has become increasingly popular as specialized ASIC designs have become more prevalent. Generators have been shown to improve the productivity of ASIC designers [13–15] and will be an important design component for specialized ASIC designs as they become commonplace. Generator-based design differs from a traditional design methodology by focusing on the development of large classes of designs rather than a single design instance. The productivity gains by this may seem counter-intuitive at first, but the ability to change significant design parameters rapidly allows designers to defer critical decision-making until more information is available.

This acceptance of late-binding changes is one of the principles in the Agile Manifesto [25], a collection of principles for improving the efficiency of software developers written in 2001. In the time since the Agile Manifesto has been written, Agile software development has become commonplace in the software development community, and its success has inspired hardware developers to follow suit. Several Agile hardware development methodologies have been developed in recent years [10, 12], which focus on improving ASIC design productivity through the use of generators and repeated full design iterations, called by the tongue-in-cheek term "Tape-in." The goal of a tape-in is to go through the full VLSI design process up to and including generation of layout data without manufacturing any masks. This process encourages the designer to automate many design flow elements and pipe cleans the generators and physical design automation used to build the chip. Tape-ins align with the Agile principle to deliver a working product frequently. Unlike software products, which are inexpensive to patch and deliver frequently, ASIC designs are very costly to manufacture, so rather than delivering the tape-in results to end customers, this process allows the designers to have a working minimum viable product (MVP) ready at all times to manufacture if needed.

Agile stands in stark contrast to waterfall development, which has traditionally been the model for the semiconductor industry. Waterfall development requires developers to freeze specifications early and pass design information to other groups, like water moving down a waterfall. This works well if the architecture and microarchitecture specifications can be frozen easily and the team is large enough, with a large enough volume of designs, to keep the pipeline of work full. However, with modern workloads like deep neural networks changing so quickly that software can become obsolete within a typical ASIC design cycle, the long latency of waterfall methodologies becomes less attractive than Agile flows. Figure 1.4 illustrates the differences in Agile and waterfall development methodologies.

1.2.4 Chipyard: An Agile generator-based SoC flow

Chipyard[7] [16] is a platform developed specifically for developing RISC-V [26] systems-on-chip using an Agile methodology. Chipyard combines many tools and intellectual property

[7]https://github.com/ucb-bar/chipyard

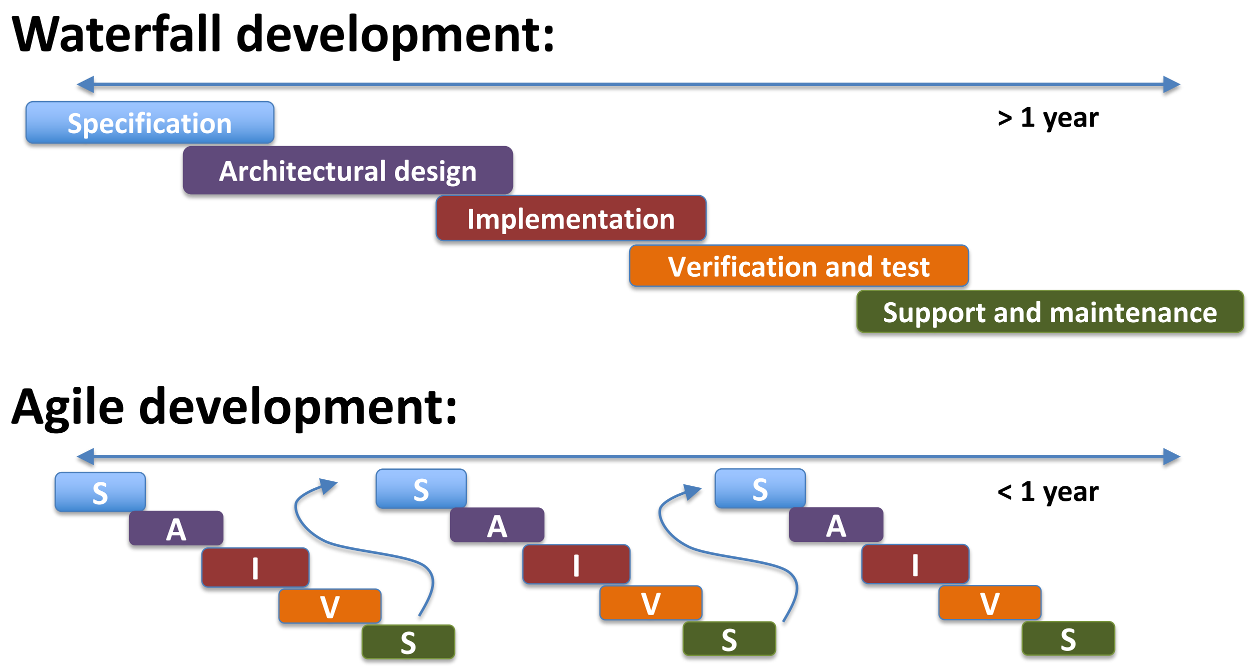


Figure 1.4: A comparison of Agile vs. waterfall development methodologies [11] (© 2018 Steven Bailey).

blocks (IP) developed at The University of California: Berkeley, along with external contributions, to provide a base SoC platform using the Rocket Chip generator [27], which includes a highly configurable in-order RISC-V CPU core, interconnect fabric, and SoC subsystem IP. Users can start with a base configuration and add custom generator IP, modify the existing generator configurations, or swap in new CPU cores. An example out-of-order core, The Berkeley Out-of-Order Machine (BOOM), is included. Chipyard supports custom IP written in Verilog, SystemVerilog, or Chisel.

Chisel [28] is a domain-specific language for constructing digital hardware embedded within the Scala programming language. Hardware construction languages differ from more traditional hardware description languages in that they are programs that call functions to generate hardware rather than being descriptions of the hardware itself. Some features of hardware construction languages are available in hardware descriptions languages. In Verilog or SystemVerilog, **generate** statements provide the user with simple construction functionality, but this is limited to basic program building blocks like numeric **for** loops or conditional blocks. Because Chisel is embedded within a modern high-level language with rich object-oriented programming support, strong typing, and powerful built-in functional programming support, users are able to build construction programs that leverage these features to build generators that are succinct and provide complex configuration options. For example, by passing functions as arguments to modules, users can inject custom hardware in a type-safe manner without messy module interface manipulation.

The current version of Chisel, chisel3, produces an intermediate representation of the generated hardware known as FIRRTL [29]. No commercial CAD tools can currently consume FIRRTL, but it is nevertheless a powerful tool that improves the usability of chisel3 over prior versions of Chisel. There are multiple levels of FIRRTL; these range from high to low and progressively become more restrictive, reducing the number of supported abstractions at lower levels. This allows advanced users and compiler writers to interact with FIRRTL at different levels of elaboration, providing the opportunity to transform or analyze the design during its compilation. The lowest level of FIRRTL is directly translatable to Verilog and avoids complex statements like **generate** to ensure support across the widest range of commercial and open-source CAD tools. The FIRRTL compiler utilizes a series of transformations or passes to elaborate the design. By default, FIRRTL preserves the cycle-level behavior of the Chisel design in the elaborated Verilog using a bare-minimum set of passes. However, there are many uses for custom transformations that modify the logical behavior of the circuit, such as introducing redundancy for resiliency or modifying the hierarchy of the design. One such transformation is the conversion of FIRRTL's memory primitive into SRAMs for use in a VLSI flow. Designing generators for an Agile flow using SRAMs can be challenging because the SRAMs usually have process- and dimension-specific module names and ports. There are also often different SRAM configurations within a process technology to support a range energy and performance criteria. By decoupling the SRAM instance information from the logical source, written in Chisel, designers can describe how to map a set of sizes to process-specific SRAM modules, converting what is effectively an $O(n \times m)$ problem into an $O(n + m)$ problem. This is further explored in Chapter 4 and Chapter 5.

In addition to RTL generation, Chipyard includes a number of additional built-in flows for other design activities, such as simulation, emulation, and physical design. A standard simulation flow exists for standard all-digital Verilog simulations with the simulator of the user's choice. More interestingly, Chipyard incorporates FireSim [30], a cloud-FPGA-based, cycle-accurate simulation engine, which cost-effectively accelerates large simulations to Megahertz-speed, a large improvement over pure software simulators. This simulation speed is crucial for the fast decision-making that is required of Agile flows.

Chipyard's physical design flow incorporates Hammer [17], a framework for creating reusable physical design components for generator-based systems-on-chip. Hammer is discussed in greater detail in Chapter 4. The Hammer framework provides tool, technology, and design abstractions to facilitate reuse of physical design collateral for generator-based flows. This reduces the effort associated with transitioning a design to a new CAD tool vendor or process technology and encourages re-use of physical design inputs for multiple designs that share common generators.

Physical design is the process of scripting and constraining the place-and-route tool[8] to implement the synthesized design. Physical design includes floorplanning, which is the placement of macros, including SRAMs, as described above, along with other physical constraints like routing and placement blockages and guides. Nearly all designs will have some amount of macro placement. Hammer provides an application programming interface (API) for creating a floorplan in a tool-agnostic format using JSON[9]- or YAML[10]-based intermediate representation, or HammerIR. Hammer floorplans are predominantly written by hand or via rudimentary scripts, but the automation of floorplans is an area of active research [18, 31]. Chapter 5 of this dissertation describes a method for generating floorplan data alongside RTL in generator-based design flows.

The tools in Chipyard adequately handle most facets of digital chip design; however, many aspects of analog or mixed-signal design are not addressed. All systems-on-chip include analog or mixed-signal IP to function, such as a phased-locked loop (PLL) for clock generation, a voltage regulator for power supply regulation, a double-data rate (DDR) memory interface, or other type of off-chip interface. The Berkeley Analog Generator (BAG) [32] is a Python framework for building process-portable analog IP generators. BAG is not integrated into Chipyard, but is used to build many of the analog and mixed signal IP used on Chipyard-based designs, such as BEAGLE [33], and the non-Chipyard designs described in Chapter 2 and Chapter 3.

To use BAG, the designer creates a schematic that connects platform-agnostic design primitives in a manner similar to traditional analog design. These design primitives are typically devices like transistors, resistors, capacitors, inductors, or diodes, but can also be more complicated groupings of devices. A set of primitive generators is then built once per process technology and re-used by all designers who generate designs on it. The layout of BAG designs is generated using a python script which utilizes BAG API calls to place the device primitives and route analog wires between them. BAG currently does not implement automatic routing like digital tools; instead, it relies on the user to provide a specific algorithm to plan the routes. The analog wires need to be larger than minimum width to reduce parasitic resistance, and designs will often have varied wire widths to optimize layout effects and parasitics. These design scripts are written so that layouts can be generated iteratively in order to converge to a design that meets given specs or to pass DRC. Nascent efforts exist to automate this process further using machine learning techniques, specifically reinforcement learning [34], and it is likely this will become more commonplace as this technology matures.

[8]In some flows, synthesis is also topologically aware and may need some physical design input.

[9]https://www.json.org

[10]https://yaml.org

1.2.5 Challenges with generator-based design

While generator-based design flows improve many aspects of the design process, there remain many unsolved challenges that are unique to such flows. Verification and floorplanning are two significant challenges among these and are of particular relevance to the integrated circuit design component of system design.

1.2.5.1 Verification

Digital chips are often verified using the Universal Verification Methodology (UVM) [35] or similar method, which relies on a scoreboard to compare transaction-level data from directed or constrained random input vectors to the output generated by the device under test (DUT). A scoreboard is effectively a module that incorporates a known-good reference model, or “Golden” model, to produce the desired output from an input stimulus. Designers rely on RTL coverage statistics, measurements of the percent of possible states wires and registers enter throughout the simulation, to gauge confidence in a design, often requiring minimum coverage metrics before allowing a design to be manufactured. This process is time consuming and only builds confidence in the specific instance being verified.

Formal verification [36] is a technique that complements constrained random verification by formally proving that user-specified properties hold for an RTL design. When used successfully, this allows verification engineers to cover cases that would take a large number of random or directed tests to verify, if even possible at all. Formal verification is more difficult to use, however, and is only successful with high quality properties and assertions. Formal verification is also only appropriate for smaller designs, as run time scales exponentially with design size.

Verification for generators currently uses these traditional verification methodologies on generated instances, which becomes a productivity bottleneck when designers are required to intervene to create new test cases or modify test benches or models for the instance under test. Generator concepts have been shown to benefit verification [37], but using verification generators still fundamentally requires re-running verification on each generated instance. Ideally the generator itself would be able to be verified, which would certify that all generated instances are correct, allowing designers to bypass the verification step for generated designs, thus increasing productivity. However, such flows have yet to be implemented and are outside the scope of this dissertation.

1.2.5.2 Floorplanning

Historically, integrated circuit floorplans have been developed manually by skilled physical designers, who work with the RTL designers to understand the data movement between components in the design. With this knowledge, physical designers choose locations for SRAM macros to optimize timing and power, often seeking to minimize the physical distance from sequential elements (SRAMs, flip flops, or latches). Physical designers must also account for electrical issues like power distribution, placement of power gating and tap cells, and placement of input-output (IO) devices that impact the die-to-package interface. More so than with verification, small changes to generator parameters can have profound effects on the floorplan of a design, especially if the change modifies the number of macros in the design. For example, even in a conventional Verilog flow, it is possible to parameterize a cache size so that a **generate** statement instantiates the required number of SRAM macros. In this flow, every parameterization of cache would require a unique floorplan. Generated instances which add or remove large amounts of placed-and-routed logic will also require floorplans which increase or decrease the available area accordingly. This issue is exacerbated by the productivity improvement afforded by generators written in high-level languages like Chisel, which allow for much more dramatic design changes with relative ease.

To add to this, floorplans are process-specific by nature, meaning a floorplan in a given process technology is not usable in another, although they may share some topology. This poses a challenge not just to generator-based designs, but to any design that a designer wants to port to a new process technology. One of the most significant impediments to process portability is the use of SRAM macros. For the same dimensions and port types, SRAM macros will have different aspect ratios and physical dimensions in different process technologies due to changing bit cell and periphery cell dimensions. While this is not always significant enough to change floorplan topology, it can be important enough to require a different macro arrangement. Furthermore, the timing characteristics of SRAMs change with feature size, shifting the optimal banking of large memory arrays and changing the number and dimensions of the SRAM macros.

1.2.6 Floorplanning concepts

In a VLSI flow, the floorplan refers to the set of physical design constraints that determine the physical arrangement of components on the die. The floorplan is typically conveyed through a series of EDA tool commands, most often using the Tcl language. Figure 1.5 shows a visualization of a simple floorplan. While most logic circuit components are placed by using an automated place-and-route algorithm, larger macrocells are manually placed by the designer in the floorplan file. These macrocells can include custom digital logic, analog or mixed-signal circuits, memories (e.g. SRAMs), or manufacturing structures like fiducial markers. Macrocell placement is the primary focus of the floorplanning effort, as it has the largest impact on quality-of-results. Placing two macrocells that interact logically too far apart creates long synchronous paths with long delays, making timing closure difficult. However, placing macrocells too densely restricts the placement of synthesized logic or signal routing between them. This is most commonly observed with SRAM cell placement, as there are typically many SRAM instances in a single digital circuit design which form logical memory banks that need to be clustered together.

Placement of top-level pins also contribute significantly to the quality-of-results produced by the place-and-route tool. For a logic block that is not a top-level chip, these pins are just metal wires to be connected at another level of hierarchy, but for full-chip designs, these pins

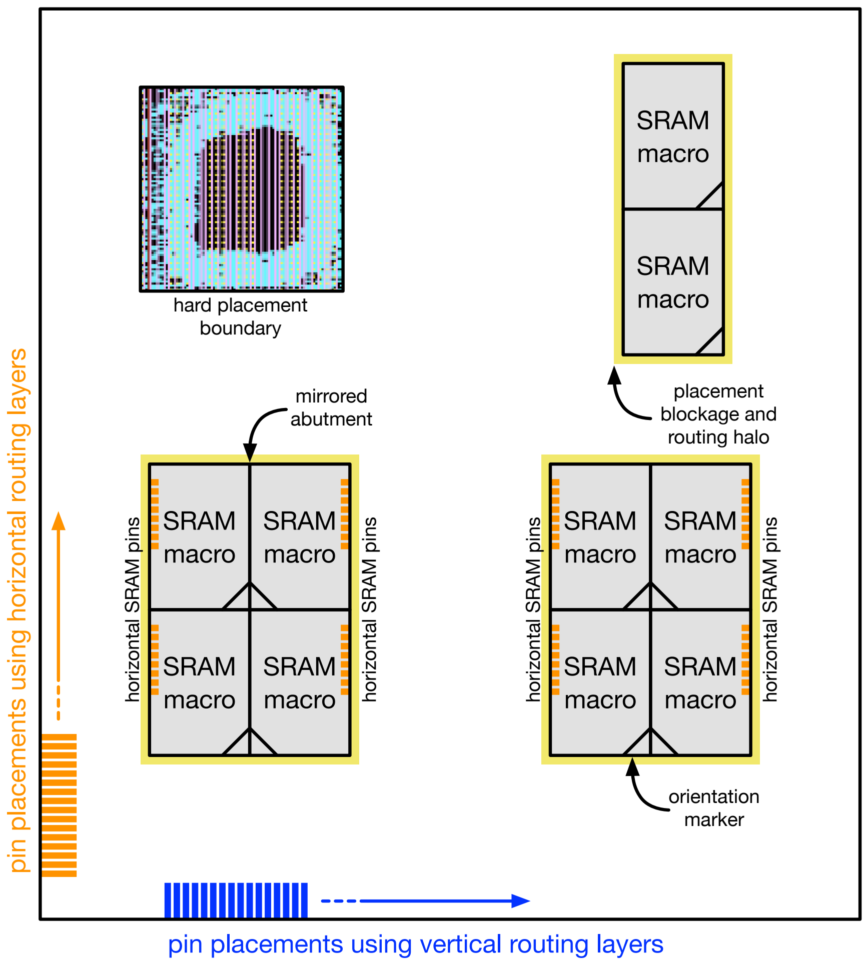
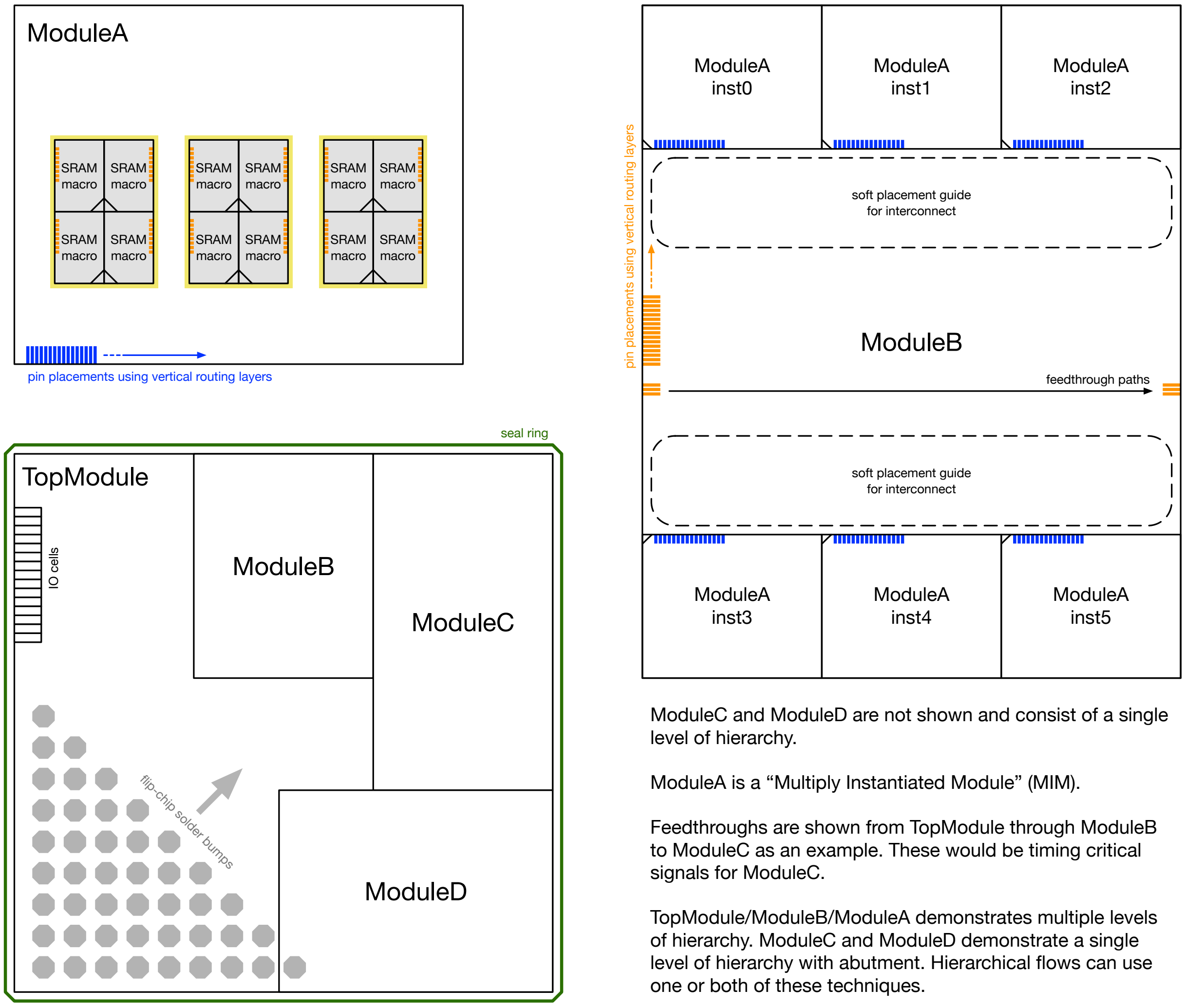


Figure 1.5: An example of a simple floorplan.

are either wire bond pads or solder bumps. Pin placement affects the coarse clustering of cells by the place-and-route tool, which must organize the design into logically connected groups based on the graph connectivity of the synthesized design. Similar to macrocell placement, pins must be grouped logically to avoid long routing delays to the relevant logic elements. For example, bits from a large bus should usually be grouped together, and if the bus has its own clock, it should be close as well. Placing pins too closely, however, regardless of connectivity, can lead to routing congestion, which manifests as a result of the routing pitch being significantly smaller than either dimension of a minimum-sized driver cell.

Beyond macrocell and pin placement, floorplans contain a number of additional physical constraints necessary to produce high-quality designs. These tend to be even more process- and design-specific than either macrocell or pin placement. They include, among others, power strap placement, routing and placement blockages, routing guides, and placement



ModuleC and ModuleD are not shown and consist of a single level of hierarchy.

ModuleA is a "Multiply Instantiated Module" (MIM).

Feedthroughs are shown from TopModule through ModuleB to ModuleC as an example. These would be timing critical signals for ModuleC.

TopModule/ModuleB/ModuleA demonstrates multiple levels of hierarchy. ModuleC and ModuleD demonstrate a single level of hierarchy with abutment. Hierarchical flows can use one or both of these techniques.

Figure 1.6: An example of a hierarchical floorplan.

boundaries. Power strap placement is discussed in Chapter 4 and is important for optimizing available routing tracks with voltage drop. Routing and placement blockages prevent the automatic place-and-route tool from using regions of the die for routing or automatic standard cell placement. This may be done to meet certain design rules, to prevent a pathological congestion issue, or for signal integrity reasons. Routing guides provide the converse effect; they encourage the tool to utilize specific routing tracks for specific routes to improve quality-of-results. Placement boundaries may be hard or soft, meaning that they are requirements or suggestions, respectively, and are guides to the tool to place logic of a specific placed-and-routed module in a given boundary. These are used in lieu of a hierarchical boundary for smaller modules that need to be isolated from other modules to meet timing, routing, or power requirements.

VLSI flows, and floorplans, are either flat or hierarchical. A flat flow means that the entire design is placed and routed entirely in a single tool run. A hierarchical flow splits the design into sub-designs which are individually synthesized, placed, and routed and subsequently incorporated into higher levels of the design. A sub-design at or above one million gates is generally considered to be a candidate for hierarchical place-and-route [38]. For a sufficiently large design, a hierarchical flow improves tool runtime and quality-of-results at the expense of flow complexity. Floorplans for hierarchical designs require knowledge of the hierarchical boundaries and require additional constraints, like pin placement for sub-blocks. Hierarchical floorplans also may include feed-through paths, which are paths that are logically unrelated to the hierarchical sub-block itself but connect neighboring elements together. An example of a hierarchical floorplan is shown in Figure 1.6.

1.2.7 State-of-the-art placement and floorplanning

Placement has been an active area of research since the earliest digital integrated circuits and continues to be an important, unsolved area of research [39]. Until recently, placement research has focused on methods of improving placement of standard cells on flat designs [40, 41] or by using hierarchical approaches [42], but automated placement of macrocells has traditionally resulted in inferior quality-of-results when compared to human-created floorplans. State-of-the-art automated mixed-size placement results as recent as 2019 [43] have been shown to be inferior to human experts [18]. Many research groups are actively investigating the applications of machine learning for physical design and floorplanning [18, 31, 44]. Of these, reinforcement learning approaches appear to have merit, with recent work showing reinforcement learning techniques can produce better quality-of-results than human experts [18]. Machine learning has also been shown to assist physical design in other areas such as IR drop estimation [45], layout parasitics and device parameter prediction [46], timing prediction [47], and clock tree prediction and optimization [48].

However, machine learning approaches have yet to become widespread, as many face steep compute costs, require large training datasets, or require rare engineers with deep technical expertise in machine learning and physical design. Many high-quality production designs continue to use human-generated floorplans, but some commercial products use generated floorplans [49]. Floorplan generators bridge the gap between fully automated placers and hand-written floorplans.

1.3 Dissertation scope and outline

While RTL generators have been shown to improve the productivity of design engineers through their expressivity, the implementation consequences of their use has not been thoroughly explored. To address this, a study of integrated circuit designs using such generators is required. This dissertation presents a series of systems-on-chip that have been designed, manufactured, and tested to demonstrate the efficacy of generator-based ASIC design methodologies, along with the methods used to create them, focusing on solutions to physical design challenges that come with generator-based design. Chapter 1 has provided the motivation for this work.

Chapter 2 and Chapter 3 describe a set of systems-on-chip manufactured in 28nm FD-SOI and 16nm finFET, respectively. This set comprises a digital ASIC spectrometer and a series of increasingly complex RISC-V systems-on-chip, starting with a single-core RISC-V system-on-chip demonstrating on-chip dynamic voltage scaling (DVS) and culminating with a 22-core heterogeneous RISC-V system-on-chip with specialized compute accelerators. Challenges faced in building each of these designs either motivate or validate components of the proposed design methodology and are listed following each chip description.

Chapter 4 describes contributions to a physical design framework tool (Hammer) inspired by some of the aforementioned systems-on-chip and used in building the others.

Chapter 5 presents a novel annotation-based floorplanning framework that leverages Hammer to ease physical design for generator-based designs by solving some of the key issues with such designs.

Chapter 6 summarizes the key contributions presented in this work and outlines future research directions.

Chapter 2

Integrated Circuit Designs in 28nm FD-SOI

A silicon-on-insulator (SOI) process technology is a planar complementary metal-oxide-semiconductor (CMOS) technology with the addition of a buried oxide (BOX), which isolates the source, drain, and channel regions from the bulk silicon underneath. In a fully-depleted SOI (FD-SOI) process technology, the semiconductor film containing the source, drain, and channel regions is thin enough that the depletion region covers the entire film. This has a number of advantages over traditional bulk CMOS, including reduced leakage and parasitic capacitance, but most importantly, this allows enables a very wide range of body biasing because the bulk silicon is isolated from the diodes in the devices. Body biasing allows circuit designers to implement both energy-efficient and high performance circuits in a single chip [50]. The integrated circuits discussed in this chapter are implemented in the ST Microelectronics 28nm ultra-thin body and BOX (UTBB) FD-SOI technology.

2.1 Splash2: Digital ASIC Spectrometer

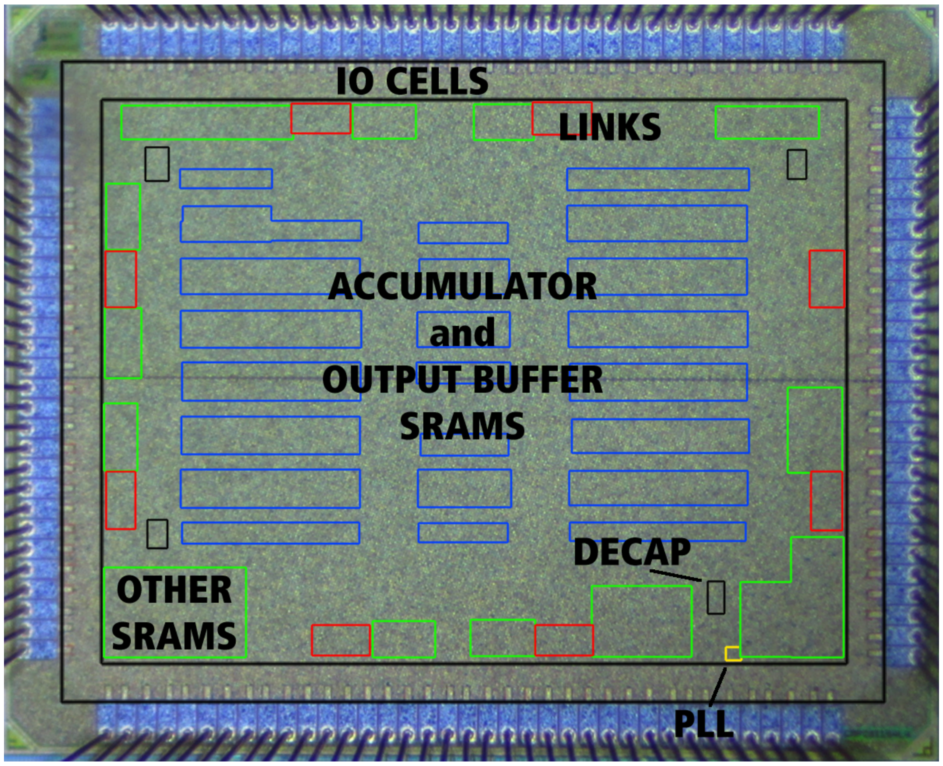


Figure 2.1: Annotated Splash2 die micrograph [51] (© 2018 IEEE).

2.1.1 Background

Earth observation satellites use digital spectrometers to measure the planet's atmospheric composition [51]. Because of the low number of total units required, these digital spectrometers are typically implemented using field programmable gate arrays (FPGAs) [52], which provide lower development cost and allow design changes in the field. However, compared to an application-specific integrated circuit (ASIC), FPGAs consume more power and are more massive. Because satellites have highly constrained power and mass budgets, ASIC solutions are more appealing than FPGAs yet are prohibitively expensive due to their low volume. However, generator-based design approaches can lower the cost of ASIC designs, making ASIC-based solutions viable.

Splash2[1] [51] is a generator-based digital ASIC spectrometer implemented in 28nm FD-SOI. The digital logic is constructed using an early version of Chisel, Chisel2, and analog components like the serial links and PLL are implemented using a standard analog design

[1]Steven Bailey led the Splash2 tapeout and testing. The author contributed the serial link transmitter design and transceiver back-end RTL and assisted with top-level integration and layout.

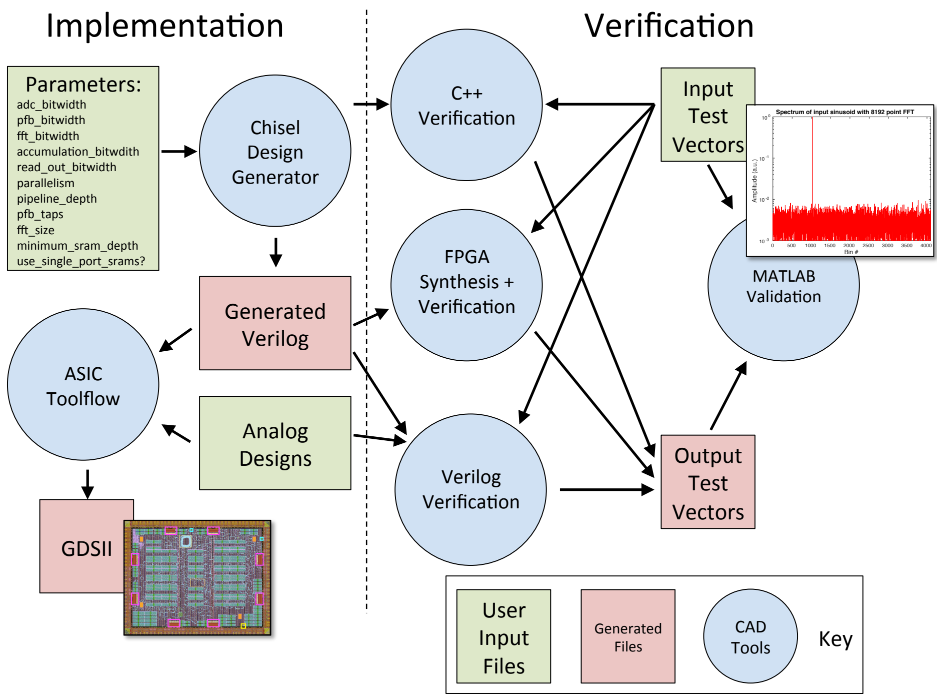


Figure 2.2: The Splash2 generator flow [51] (© 2018 IEEE).

flow. An annotated die micrograph of Splash2 is shown in Figure 2.1, and the generator development flow is shown in Figure 2.2. The ASIC is verified by generating directed test vectors and using a Matlab model to verify the correctness of the output spectrum. The source Chisel code is simulated using a C++ simulator[2], while the generated Verilog is verified via FPGA implementation and through direct simulation. The ASIC physical design flow uses a standard reference methodology with Synopsys EDA tools.

,

2.1.2 Architecture

Splash2 computes the spectrum of a signed 4-bit data stream. This data can either be generated by an internal test vector generator (TVG), used for bring-up and testing, or provided via 8 high-speed serial links [53], shown in Figure 2.3. When provided over the high-speed serial links, the input data format can be mapped using a runtime-programmable 4x4 look-up table (LUT), allowing compatibility with different external components. The

[2]The C++ simulator back-end is no longer supported in modern Chisel releases (3.0+), which instead encourages Verilog simulation using a Verilator[3] or a proprietary simulator. Alternatively, Treadle[4] can be used to simulate FIRRTL directly.

[3]https://veripool.org/verilator/

[4]https://github.com/chipsalliance/treadle

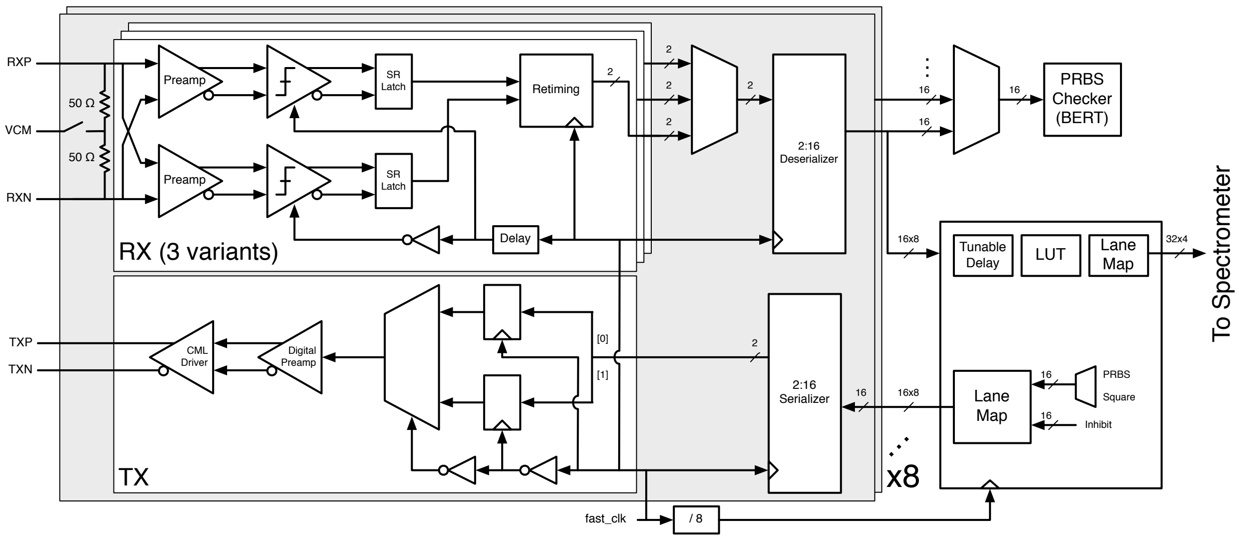


Figure 2.3: The Splash2 ADC interface using serial links [51] (© 2018 IEEE).

serial links use an external calibration loop to adjust the clock phase, which is configured each time the system is powered on.

Figure 2.4 shows the connection between the Splash2 die and an external analog-to-digital converter (ADC), which provides an unsigned 3-bit data stream plus over/under-range bit. Two serial link transmitters are used to provide an XOR modulation signal and an inhibit signal to the ADC. The XOR signal is driven by a pseudo-random binary sequence (PRBS) circuit, which is used to generate edges in the data stream for approximate DC balance and clock recovery. The inhibit signal prevents the ADC from producing data, which effectively propagates the XOR signal to the outputs X and Y. This allows the Splash2 die to align PRBS circuits in each data channel for demodulating the XOR signal from the data stream once inhibit is deasserted. Data is interleaved between the two data channels X and Y to reduce the necessary serial link frequency, with the channels offset by half a unit interval (UI).

The microarchitecture of the Splash2 digital spectrometer is shown in Figure 2.5. The spectrometer input is selected from either the ADC interface or TVG and fed into a 4-tap polyphase finite impulse response (FIR) filter bank (PFB) with 8-bit coefficients. The PFBs are used to condition the data before it is sent to the fast Fourier transform (FFT) logic in 8-bit format. PFB coefficients are compressed using delta compression, reducing the coefficient storage from 32 KiB to 8 KiB.

The FFT instance is an 8192-point split architecture, with parallel biplex FFTs driving a direct FFT to reduce hardware overhead, as shown in Figure 2.6. The FFT outputs thirty-two 16-bit real and thirty-two 16-bit imaginary data per cycle. Synthesis retimes the two pipeline stages in each FFT stage to optimize the critical path.

The FFT drives 64-bit accumulators, which accumulate over a programmable number of spectra. Once accumulated, the data is stored in an SRAM buffer for slow read-out through

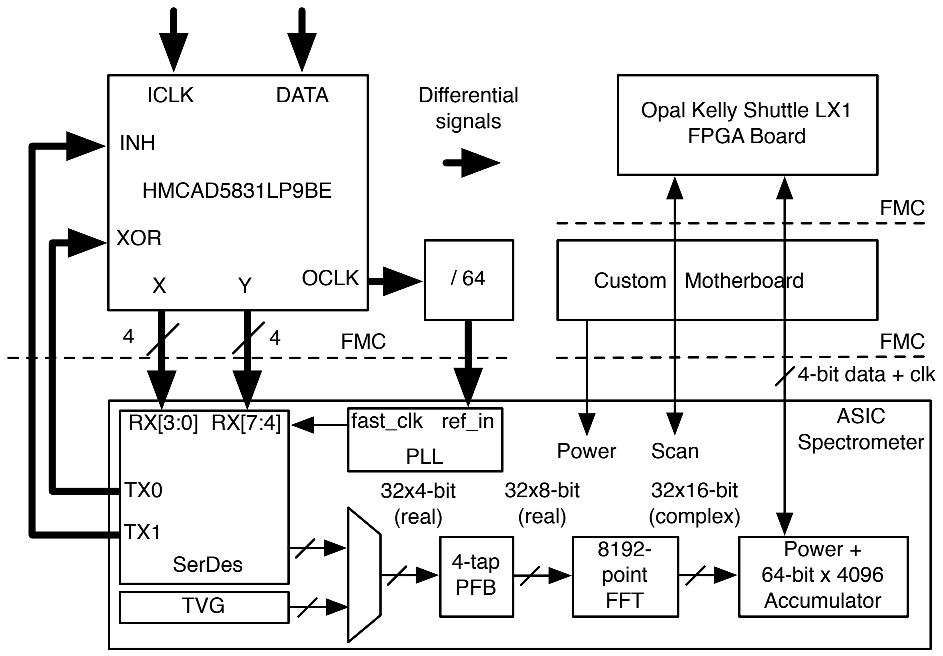


Figure 2.4: The Splash2 system block diagram [51] (© 2018 IEEE).

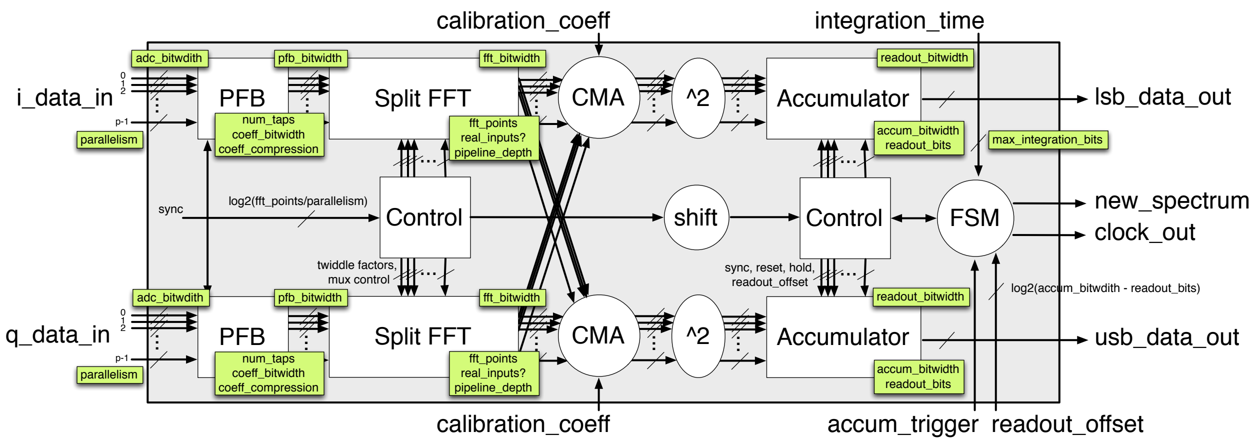


Figure 2.5: The Splash2 spectrometer generator microarchitecture [51] (© 2018 IEEE).

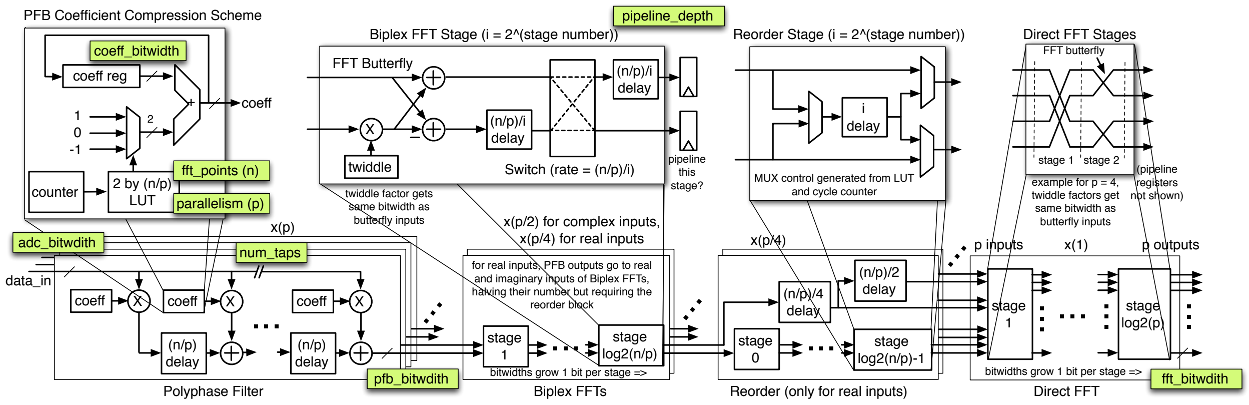

PFB Coefficient Compression Scheme
coeff_bitwidth
coeff reg
coeff
1
0
-1
2
counter
2 by (n/p) LUT
fft_points (n)
parallelism (p)
pipeline_depth
Biplex FFT Stage (i = 2^(stage number))
FFT Butterfly
X
twiddle
(n/p)/i delay
(n/p)/i delay
Switch (rate = (n/p)/i)
pipeline this stage?
twiddle factor gets same bitwidth as butterfly inputs
x(p/2) for complex inputs, x(p/4) for real inputs
Reorder Stage (i = 2^(stage number))
i delay
MUX control generated from LUT and cycle counter
Direct FFT Stages
FFT butterfly
stage 1
stage 2
example for p = 4, twiddle factors get same bitwidth as butterfly inputs
(pipeline registers not shown)
adc_bitwdith
x(p)
num_taps
data_in
coeff
coeff
coeff
(n/p) delay
(n/p) delay
pfb_bitwdith
Polyphase Filter
for real inputs, PFB outputs go to real and imaginary inputs of Biplex FFTs, halving their number but requiring the reorder block
stage 1
stage log2(n/p)
bitwidths grow 1 bit per stage =>
Biplex FFTs
x(p/4)
(n/p)/4 delay
(n/p)/2 delay
stage 0
stage log2(n/p)-1
Reorder (only for real inputs)
p inputs
x(1)
p outputs
stage 1
stage log2(p)
bitwidths grow 1 bit per stage =>
fft_bitwdith
Direct FFT

Figure 2.6: The Splash2 polyphase filter and streaming FFT generator microarchitecture [51] (© 2018 IEEE).

the chip I/O, which occurs in parallel with the accumulation of the next result.

2.1.3 Results

The Splash2 die is fabricated in 28nm FD-SOI, measuring 1.8 mm by 2.3 mm. The serial link receivers function with a bit error rate (BER) below 10^{-7} at 5 GHz double-data rate (DDR) using a novel modified StrongARM architecture [54]. When using two separate 4-bit channels, this corresponds to a maximum ADC frequency of 20 GS/S or a 10 GHz Nyquist bandwidth. Using the internal TVG, the spectrometer logic functions up to 530 MHz, which corresponds to 17 GS/s or an 8.5 GHz Nyquist bandwidth. The overall system is limited to 1.5 GS/s due to clock noise injected into the serial link receivers when both circuits are powered on. The Splash2 ASIC dissipates 1.0 W at 530 MHz and the ADC dissipates 4.2 W.

Figure 2.7 plots an unaccumulated output spectrum alongside an accumulation of 800 spectra, demonstrating both the correctness of the ASIC and the reduction in noise power with longer accumulations. Table 2.1 compares Splash2 with state-of-the art ASIC spectrometers at the time of its publication.

2.1.4 Physical design challenges

The Splash2 ASIC contains a large amount of on-chip SRAM which is difficult to place optimally. Figure 2.8 shows the floorplan of the ASIC. The cross-hatched rectangles represent macrocell placements, while the amorphous shapes surrounding them indicate areas of standard cells which have been placed and routed by the EDA tool. With the exception of the serial links, PLL, clock receivers, I/O cells, decoupling capacitors (decap), and fiducial markers, which are annotated in the figure, all macrocells in this design are SRAMs. The large number and varied dimensions of SRAMs in this design relative to its size require a custom floorplan. Given that Splash2 is an instance of a generator with a wide parameterization space, the creation of this custom floorplan requires additional engineering overhead

Table 2.1: Splash2 compared with state-of-the-art ASIC spectrometers [51] (© 2018 IEEE).

Reference	Splash2 [51]	CICC '15 [55]	CICC '09 [56]
Technology	28nm FD-SOI	65nm CMOS	90nm CMOS
Bandwidth	**8.5 GHz**	1.1 GHz	0.75 GHz
FFT Size	**8192 pts**	512 pts	**8192 pts**
Integrated ADC	No	**Yes**	No
Total Power	5200 mW	**188 mW**	1500 mW†
FOM (pts·GHz)/mW	**13.4**	3.0	4.1

†Excludes ADC

A bold cell indicates the best metric in each category.

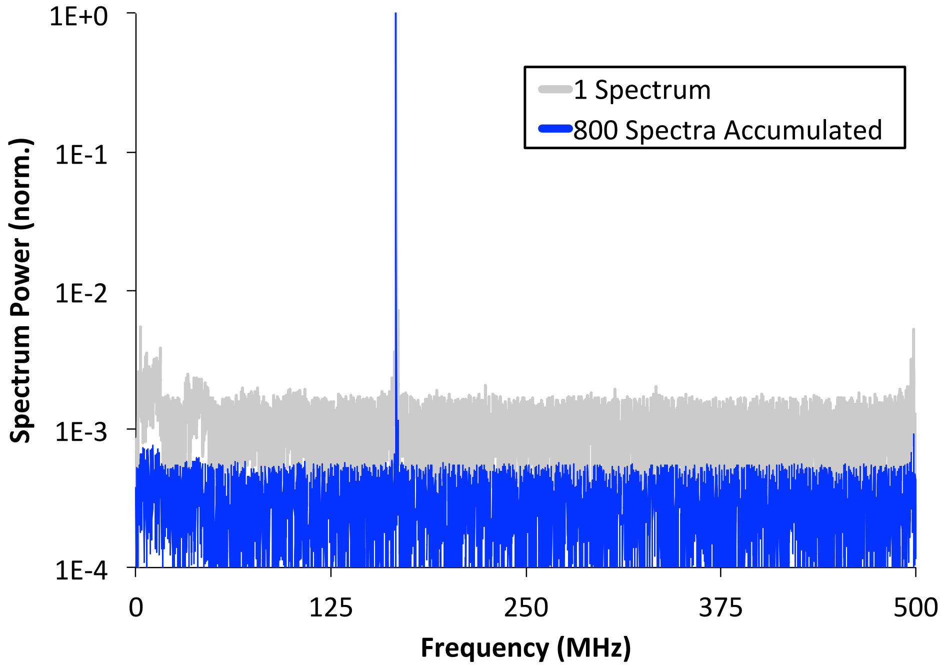


Figure 2.7: Splash2 measured spectra at a 1 GHz sample frequency [51] (© 2018 IEEE).

for each generated instance that is linear with the number of instances created. For example, changing the number of points in the FFT or the size of the output buffer would change the dimension and counts of the SRAMs present in the design. This overhead would be dramatically reduced with a floorplan generator.

A second significant physical design challenge with the Splash2 ASIC is the placement of the twiddle factor ROM, which is an array of constant coefficients used in the FFT algorithm. This ROM is implemented using combinational logic gates, which creates a logic circuit with high wiring density. Without intervention, this causes significant routing congestion during the route step. A solution to this problem is the creation of a hard boundary for the twiddle factor ROM module which is larger than the area of the otherwise unconstrained ROM module, which allows the placement tool to relax placement to decrease wiring density. An alternate solution is the use of a hardened ROM macro, which is similar in architecture to an SRAM but with constant bit cells, but this solution is not generalizable to all instances of combinational logic with similar routing issues. A floorplan generation platform must therefore support this type of placement constraint.

All eight serial links back-ends on Splash2 are synchronous to each other because of the requirement to assemble four bits of data striped across four lanes into a single word. This makes timing closure difficult, as the clock skew across the entire die, even a relatively small die, can be substantial. A better solution is to place the serial links closer together or to use an ADC which outputs serial data rather than parallel data, which would allow the serial links to operate in their own asynchronous clock domains.

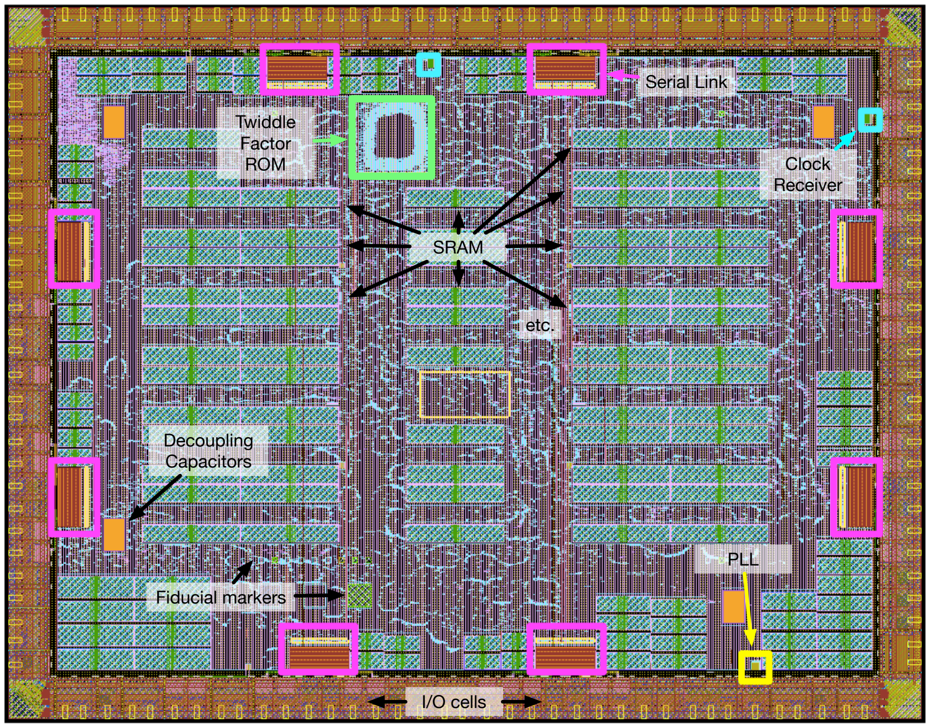


Figure 2.8: Splash2 floorplan showing the highly congested twiddle factor ROM and SRAM macro placement.

2.2 Hurricane1: A Dual-Core RISC-V SoC with DVS

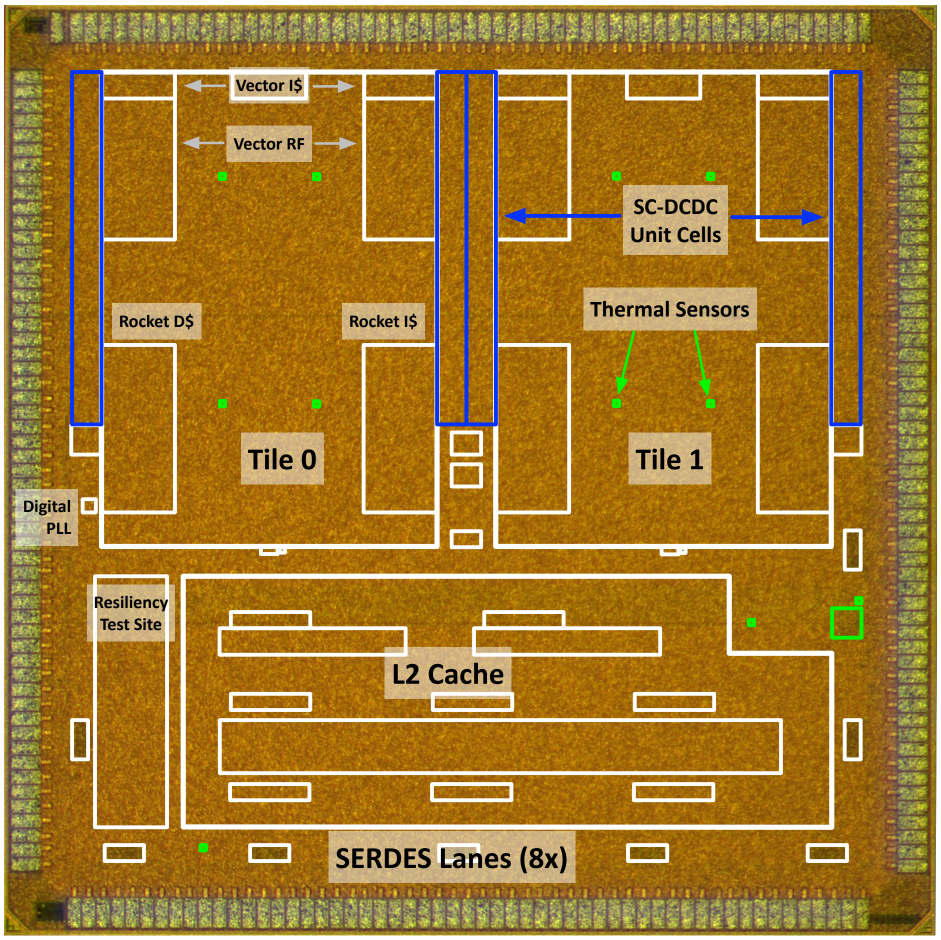


Figure 2.9: Annotated Hurricane1 die micrograph [57] (© 2020 IEEE).

2.2.1 Background

Consumers of modern technology have grown accustomed to steady improvements in performance and functionality with each product release cycle without having to sacrifice battery life. Battery life is a driving factor in device purchases, e.g. smart phones [58], because needing to charge a device more than once a day, typically while its owner is sleeping, is a major inconvenience. Absent a major advancement in battery technology, this imposes a restriction on device power consumption. This is a challenge for system-on-chip designers, who must continue to improve compute performance for edge applications while staying within a fixed power envelope.

The end of Dennard scaling [3] has reduced the ability to continue making performance and power improvements by simply waiting for the next advancement in process technology. Instead, the incremental power cost of increasing circuit performance must be offset using architectural or circuit-level techniques, with the end goal of reducing energy per task irrespective of peak power. Dynamic voltage scaling (DVS) [59] is a technique to improve energy efficiency in digital systems. The use of DVS translates an approximately linear reduction in system performance into quadratic savings in switching energy and more-than-quadratic savings in leakage energy, significantly lowering the energy per task as a result. Use of DVS during periods of low application performance allows compute demand to be met at lower energy operating points. This has been shown to improve overall system efficiency when used in a system with a bounded throughput requirement [60].

The overall energy efficiency of a DVS system depends on both the conversion efficiency of the voltage regulators and the ability of the system to predict and to react to upcoming changes in the workload. An ideal DVS system transitions as quickly as possible when a workload changes its compute performance requirements, as energy is wasted for the duration of time the system remains in a sub-optimal DVS state.

DVS can also be applied to multiple regions on a chip simultaneously. Recent mainstream products use platform-level regulators and per-core integrated low dropout linear regulators (LDOs) to achieve 19% power savings [61]. In typical DVS applications, the voltage regulation is performed off-chip. The number regions, or voltage-frequency domains, is therefore limited by the available number of distinct supply connections allowed by the chip pinout and packaging strategy. An alternate approach is to include the voltage regulators on-chip, but this can be costly owing to the large amount of area necessary for passive devices like capacitors and inductors. One approach that mitigates this issue is the inclusion of passives in the package [62]. An alternate approach foregoes a fixed direct current (DC) supply to obviate the need for expensive on-die capacitors by using rippling, on-die switched-capacitor DC-DC converters [63]. Voltage scaling can also be augmented with circuit-level error-detection techniques to recover supply voltage margin [64].

Hurricane1[2] [57], shown in Figure 2.9, is a dual-core RISC-V system-on-chip implemented in 28nm FD-SOI with integrated on-die DC-DC voltage converters and a high-speed serial-link memory interface. The digital logic is constructed using Chisel2, as described in Section 2.1, and analog components like the serial links and PLL are implemented using standard analog design flow.

2.2.2 Architecture

[3] Hurricane1, shown in Figure 2.10, comprises a dual-core, 64-bit RISC-V processor with a custom vector accelerator, manufactured in a 28nm fully-depleted silicon-on-insulator (FD-SOI) technology. Hurricane1 is the first reported RISC- V multi-core design to place each core in its own voltage domain and utilize high-speed serial links for memory traffic. The supply voltage for each core is provided by a bank of 24 on-chip switched-capacitor DC-DC

[2]Benjamin Keller led the Hurricane1 tapeout. Others contributed the DC-DC converter design and vector accelerator design. The author contributed the serial link transmitter design and transceiver back-end RTL; assisted with system-level RTL, integration, layout; and led testing for the manuscript.

[3]This section contains text that is © 2020 IEEE [57].

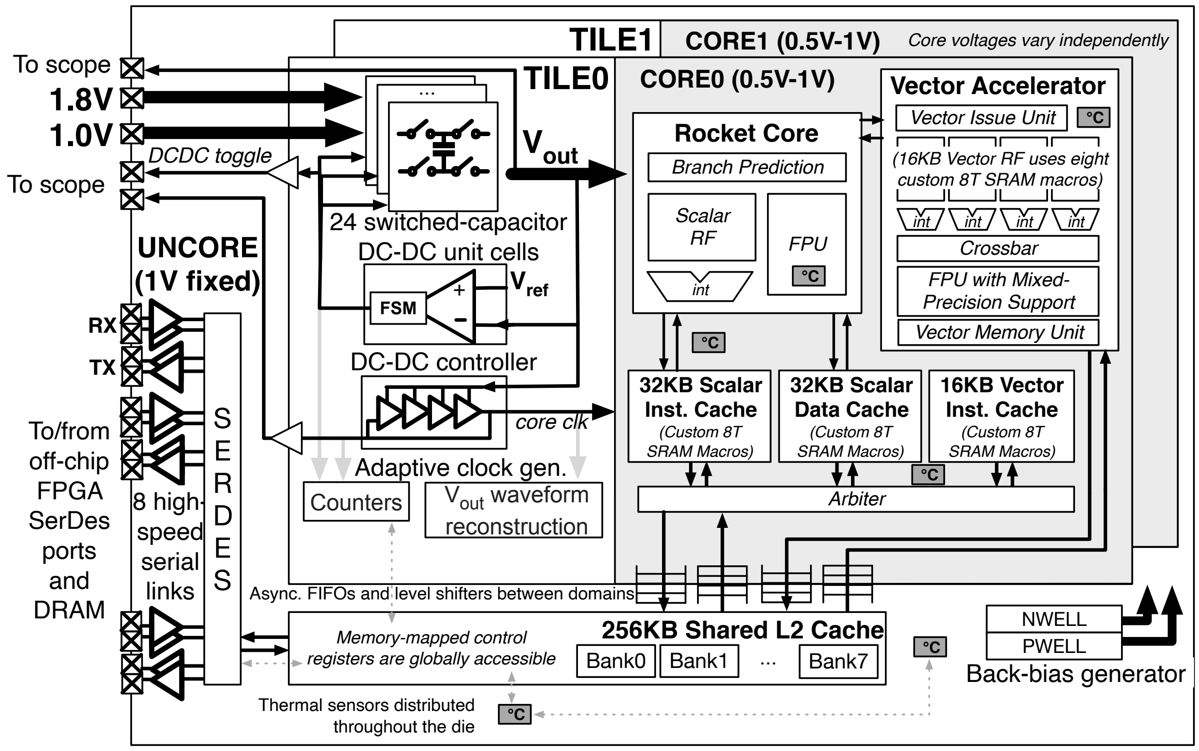

TILE1
CORE1 (0.5V-1V)
Core voltages vary independently
TILE0
CORE0 (0.5V-1V)
To scope
1.8V
1.0V
DCDC toggle
To scope
V_{out}
24 switched-capacitor DC-DC unit cells
FSM
V_{ref}
DC-DC controller
core clk
Adaptive clock gen.
Counters
V_{out} waveform reconstruction
UNCORE (1V fixed)
RX
TX
To/from off-chip FPGA SerDes ports and DRAM
8 high-speed serial links
SERDES
Rocket Core
Branch Prediction
Scalar RF
FPU
int
°C
Vector Accelerator
Vector Issue Unit
(16KB Vector RF uses eight custom 8T SRAM macros)
Crossbar
FPU with Mixed-Precision Support
Vector Memory Unit
32KB Scalar Inst. Cache (Custom 8T SRAM Macros)
32KB Scalar Data Cache (Custom 8T SRAM Macros)
16KB Vector Inst. Cache (Custom 8T SRAM Macros)
Arbiter
Async. FIFOs and level shifters between domains
256KB Shared L2 Cache
Memory-mapped control registers are globally accessible
Bank0
Bank1
Bank7
NWELL
PWELL
Back-bias generator
Thermal sensors distributed throughout the die

Figure 2.10: Hurricane1 block diagram [57] (© 2020 IEEE).

converter cells [60] which can be bypassed if needed. This system includes the same DC-DC conversion subsystem as [65], which is capable of switching modes within 2 µs. Each core can be clocked by an externally-supplied clock or an on-chip adaptive clock generator [66]. The clock selection is made by writing the select value to a memory-mapped register using the off-chip memory interface while the cores are in reset. The two cores share a 256KiB L2 cache, which is in a separate, fixed voltage and clock domain [60]. The L2 cache is also responsible for routing memory traffic to memory-mapped control and status registers. It includes a set of counters that track how many total memory accesses have hit and missed. The L2 cache has two possible paths for backing memory: a custom, low-speed, eight-bit parallel interface or a bank of eight high-speed serial links. Consistent with our goal of innovating at the circuit level while maintaining a functional system, the parallel interface is included as a backup to the experimental high-speed interface. Each of these paths forwards memory transactions to a separate FPGA board which contains the backing DRAM for the system. The system also contains multiple, distributed ring-oscillator-based temperature sensors [67] and a body-bias generator [68].

The Rocket applications processors are scalar, in-order, single-issue cores with five pipeline stages [27]. This version of Rocket supports version 2.1-draft of the RISC-V RV64G ISA variant with supervisor mode. These Rocket instances were generated from param-

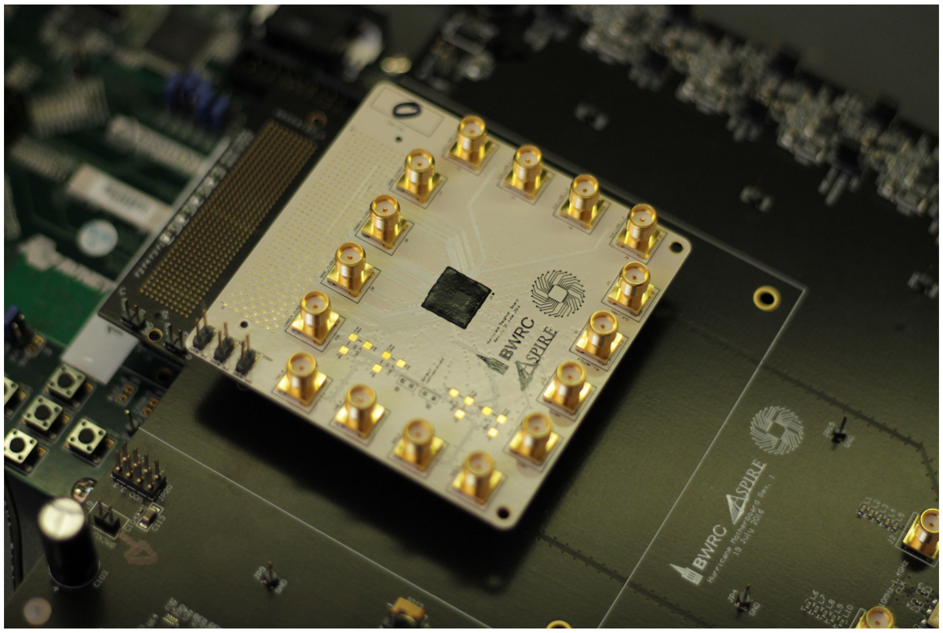


Figure 2.11: Hurricane1 board photo.

eters chosen for a typical in-order, general-purpose core. It has a 64-entry branch target buffer, a 256-entry two-level branch predictor, a return address stack, a two-cycle latency on single-precision fused multiply-add (FMAs), a three-cycle latency on double-precision FMAs, and separate 32KiB instruction and data caches. The Rocket core contains a set of performance counters that track how many instructions have been retired and the number of cycles executed.

The custom vector accelerator, Hwacha [69], is a configurable, multi-lane decoupled vector pipeline optimized for an ASIC process that executes the Hwacha ISA version 3.8.1 [70]. In this design, each core is configured with a single-lane variant. Hwacha more closely resembles traditional Cray [71] vector pipelines than the SIMD units in SSE or AVX [72]. Hwacha improves efficiency by offloading vector instructions from the scalar core to the vector unit, allowing the scalar core to continue to execute in parallel and enabling the vector unit to prefetch its own data from memory [73] effectively. Hwacha is connected to the scalar core via the Rocket Custom Coprocessor (RoCC) interface [27]. This instance of Hwacha includes four banks of 256x128 dual-port SRAMs for the vector register file, per-bank integer ALUs, four double-precision FMA units, eight single-precision FMA units, sixteen half-precision FMA units, eight master sequencer slots, a 16KiB vector instruction cache, and a single 128-bit wide port to the L2.

The cores operate independently and are cache-coherent. The SoC can execute programs with multiple concurrently running threads with each core capable of performing its own power management, which is done by setting its DC-DC configuration register to one of three switching modes: 1/2 1V, 2/3 1V, or 1/2 1.8V [65]. These registers are memory-mapped

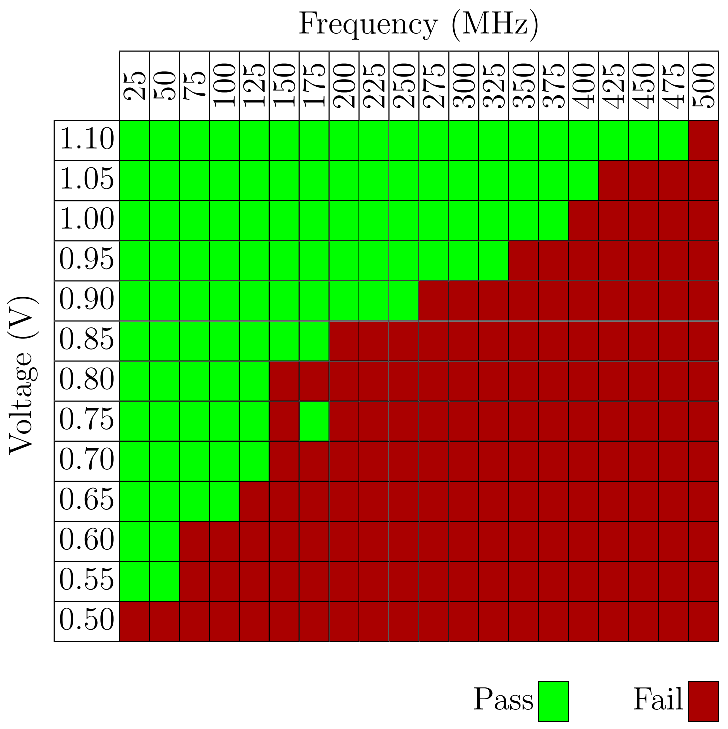


Figure 2.12: Hurricane1 DGEMM shmoo plot [57] (© 2020 IEEE).

and accessible by either core or the external memory interface. When adaptive clocking [66] is enabled, the clock will automatically transition along with the changing voltage profile.

It is possible to write customized power management code for a specific application, but this either requires building a specialized toolchain or embedding power management code into the application code. A more productive approach is to use the standard, open-source RISC-V toolchain to build the application and provide autonomous power management code separately. To demonstrate such autonomous power management in this system, one core is used as an applications core and the second core is programmed to act as a power management unit (PMU) for the first. The PMU can execute independent power management code which monitors the performance and power utilization of the system by reading a set of memory-mapped counters and control registers.

Switched-capacitor DC-DC converters offer the ability to perform direct power measurements by monitoring the toggle rate of the switches within the converter [65]. In a contrasting approach used by this work, architectural counters offer a prediction of near-future workloads [62, 74–76]. The set of counters implemented in this chip enables power management by measuring the short-term off-chip memory bandwidth used by the L2 cache, the rate of instruction execution, and the rate of energy consumption by the DC-DC converters. Prior work has identified multiple classes of counters to observe the state of each core, queues, and the memory system [62, 75]. This work utilizes cache miss counters to provide insight into near-term core activity, as a cache miss will cause a core to be idle for hundreds of cycles.

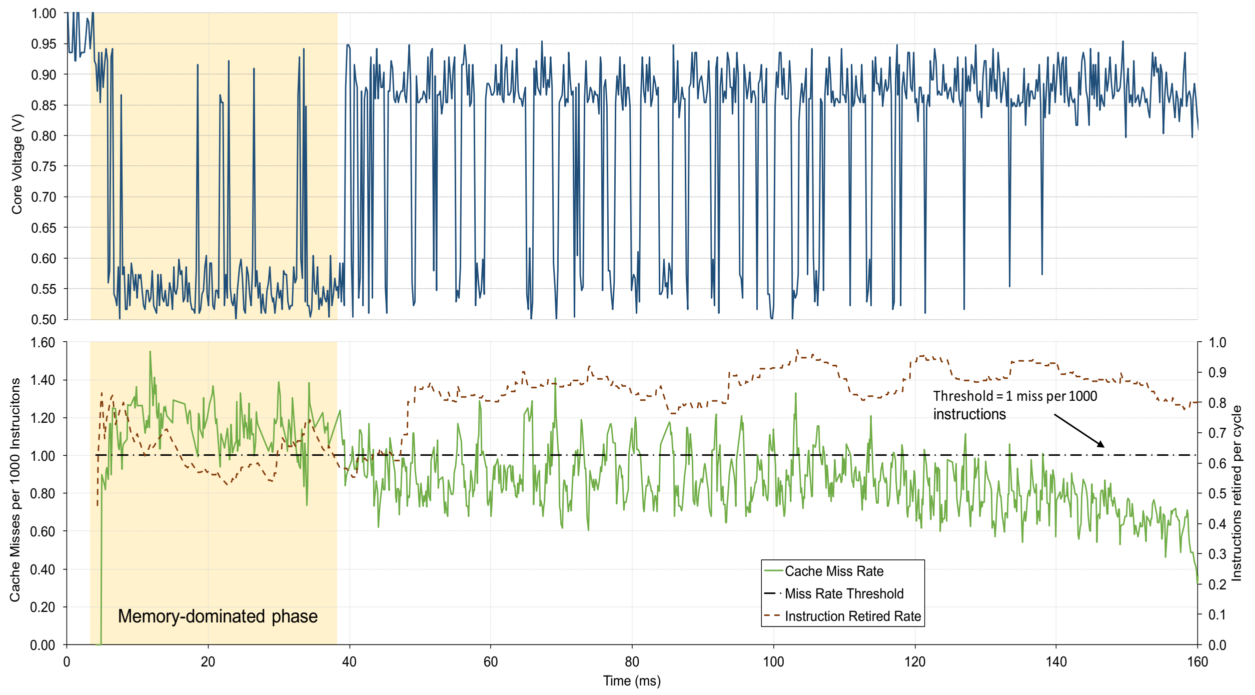


Figure 2.13: Hurricane1 DC-DC mode transitions in response to cache activity for a matrix multiplication workload [57] (© 2020 IEEE).

Therefore, by using on-chip DC-DC converters with small response times, the core supply can be lowered until the data is retrieved without losing state, after which the voltage can be quickly increased to achieve the desired throughput. Although not demonstrated in this work, other energy-saving techniques, such as reverse body biasing [68], can also be actuated in a similar manner.

The PMU is programmable using standard C with header files that define the locations of the performance and system monitor counters. The fully-featured PMU architecture enables many experiments and studies on effective power management strategies in this system. However, a more production-level system might instead include a tiny, feature-reduced core as a dedicated PMU or, alternatively, use the OS or interrupts to run power management code on one of the applications cores only a fraction of the time.

2.2.3 Results

Hurricane1 is fabricated in 28nm FD-SOI, measuring 2.8mm by 2.8mm. To run tests, the chip is connected to an FPGA board for tethering, control, and backing memory. A photo of the test setup is included in Figure 2.11. The chip is capable of running complex workloads including booting Linux and running applications under operating-system support, through both the slow parallel interface and a high-speed link. This system includes many features of embedded systems that enable it to be an effective test platform for fine-grain adaptive

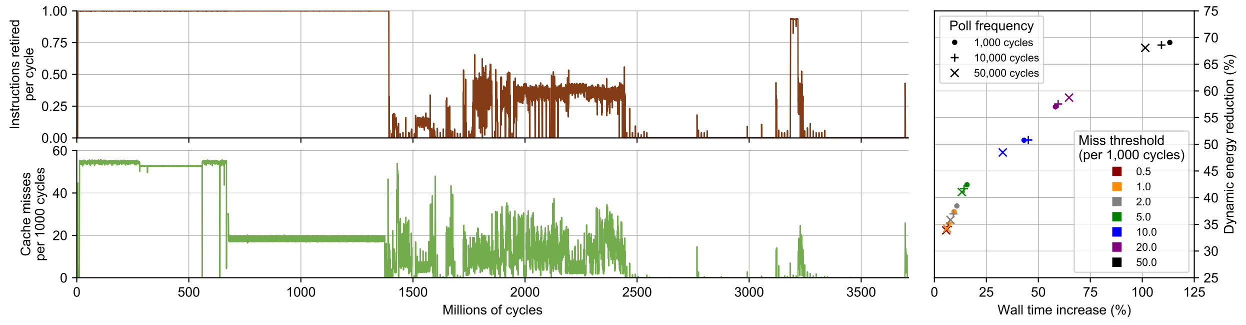


Figure 2.14: Hurricane1 architectural simulation of Linux boot and energy analysis at 0.9V/250MHz and 0.55V/50MHz operating modes [57] (© 2020 IEEE).

dynamic voltage and frequency scaling experiments. Table 2.2 demonstrates the efficacy of this system compared to prior work.

The chip is tested by running a double-precision matrix multiplication and verifying correctness while sweeping voltage and frequency to determine the optimal operating points for the available power states. Figure 2.12 displays the results of the sweep of operating points, outlining the optimal voltage-frequency curve. The non-monotonic behavior around 175 MHz is attributed to a small supply resonance observed on-die.

The most energy-efficient operating point at which the system operates error-free is 525 mV at 28.3 MHz and results in a core energy efficiency of 19.6 GFLOPS/W[4]. When compared to our previous work [65], this work is less efficient in this application because the cores are not clock gated and its larger die requires slightly higher minimum operating voltage. [5] This operating mode uses the second, full-featured core as a PMU. This core contains inactive hardware like vector and floating-point units and is clocked at a fixed frequency; it is therefore a pessimistic efficiency model for a comparable single-core system.

The effectiveness of the fully-programmable PMU is demonstrated with a program that monitors the rate of L2 cache misses and the average number of instructions-retired-per-cycle (IPC) of the compute core, changing the voltage mode of the compute core in response, while keeping the L2 supply constant. In this experiment, the frequency is held constant, although it is possible to implement adaptive clocking as in previous designs [65]. The PMU polls these counters every 100 µs and records their values while the voltage of the compute core is monitored with an external oscilloscope, as shown in Figure 2.13. Each time the counter is polled, the current and prior cumulative cache miss counts are subtracted to determine the miss rate. When the program detects fewer than one cache miss per 1000 cycles, it infers that the compute core is in a compute-bound section of the application and thus increases the voltage mode to the maximum, 900 mV. Otherwise, the program infers that the compute core is in a memory-bound section and reduces the operating voltage.

Figure 2.13 illustrates this PMU program executing mode transitions while the compute core performs a matrix multiplication, sized to fit in the L2 cache. This compute kernel is

[5]1 GFLOPS = 1 Giga (10^9) floating point operation per second.

Table 2.2: Hurricane1 compared with prior art [57] (© 2020 IEEE).

Reference	Hurricane1 [57]	JSSC '17 [77]	IBM JRD '15 [78]	Hot Chips '11 [79]
Technology	28nm FD-SOI	28nm FD-SOI	22nm SOI	32nm
Off-chip components	No	No	Yes	Yes
AVS response time	< 2 µs	< 2µs	32µs	≈ 1000µs
Workload-dependent DVS algorithm	Yes	No	Yes	Yes
Energy efficiency (GFLOPS/W)	19.6 DP	41.8 DP	-	3 DP*

*Estimated.

chosen to highlight two distinct program phases: A memory-dominated phase arising from compulsory cache misses at the start of the program and a compute-driven phase after the L2 is full. The behavior of workloads with more complex memory access patterns can be extrapolated from the phases in this example.

The threshold of one miss per 1000 cycles has been set empirically for this application but, given its programmable nature, can be tuned for different workloads. Using FireSim [30], an architectural simulation of a similarly-configured core demonstrates this technique during Linux boot, shown in Figure 2.14. While this first stage bootloader implementation spins in a loop, causing IPC to stay near 1, the subsequent activity is shown to benefit from DVS. From the simulated cache miss counter data and measured voltage and frequency data, the relative changes to wall time, $\Delta t_{\%}$, and dynamic energy, $\Delta E_{\%}$, are modeled with the following formulas. The time spent in the low voltage and frequency state, $cycles_{lo,rel}$, is calculated relative to the cycles in the high state, $cycles_{hi}$, by scaling the instructions retired per cycle when the miss rate is below the threshold, IPC_{lo}, by the ratio of high and low frequencies. This calculation is shown in equation 2.1, equation 2.2, and equation 2.3.

$$cycles_{lo,rel} = IPC_{lo} \cdot cycles_{lo} \cdot \frac{f_{hi}}{f_{lo}} + (1 - IPC_{lo}) \cdot cycles_{lo} \tag{2.1}$$

$$\Delta t_{\%} = \frac{cycles_{hi} + cycles_{lo,rel}}{cycles_{hi} + cycles_{lo}} - 100\% \tag{2.2}$$

$$\Delta E_{\%} = 100\% - \frac{cycles_{hi} + cycles_{lo,eff} \cdot (\frac{vdd_{lo}}{vdd_{hi}})^2 \cdot \frac{f_{lo}}{f_{hi}}}{cycles_{hi} + cycles_{lo}} \tag{2.3}$$

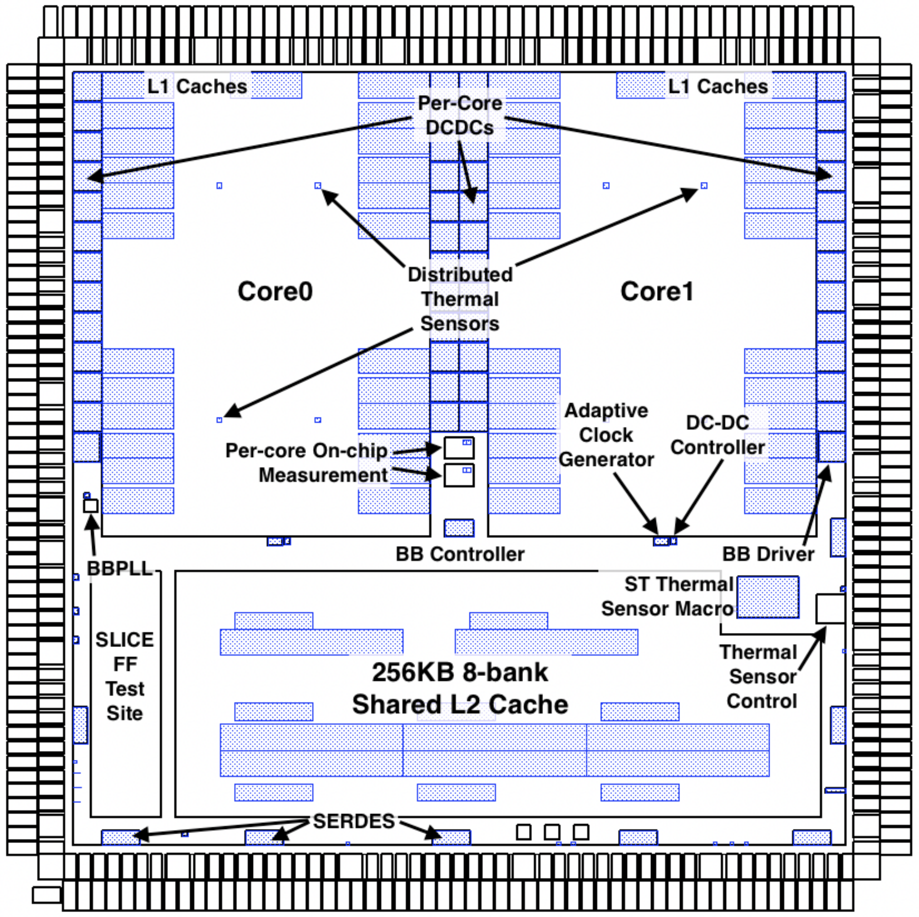


Figure 2.15: Annotated Hurricane1 floorplan [80] (© 2017 Benjamin Keller).

A range of thresholds and polling frequencies, shown in Figure 2.14, allows for control between a 69% relative energy savings for 113% additional wall time and a less aggressive 34% energy savings for only 6% additional wall time. By increasing the miss rate threshold, the voltage state is changed less frequently and only when more or larger memory accesses are performed, leading to an increase in idle time while the core voltage is high. Decreasing the miss rate threshold causes the voltage state to be more sensitive to memory activity, which may increase the non-idle time spent at the lower voltage state, thus hurting performance.

2.2.4 Physical design challenges

At 7.84 mm^2, the Hurricane1 die is 89% larger than the Splash2 die at 4.14 mm^2. For a flat design, additional area increases tool runtime superlinearly. Hurricane1 contains two identical CPU cores, which make it a good candidate for hierarchical design, as this can be used to divide the design into three roughly equally-sized areas. Hierarchical design therefore reduces the size of the design implemented in a single tool flow to one that is smaller than

Splash2.

However, this Hierarchical design methodology adds additional tool flow complexity which is exacerbated by the use of Chisel2 for RTL generation. In a Verilog or SystemVerilog design, it is known whether or not a particular hierarchical instance is modified in each design iteration, which allows designers to re-implement only the modified design components to save tool runtime. In a generated design, and in particular one using Chisel2, it is not always easy or possible to modify the design in a way that deliberately preserves a given hierarchical module. This means that the entire tool flow must be re-run for a given RTL change or the user must manually run logical equivalency checking to guarantee a hierarchical cell has not changed between physical design iterations.

The Hurricane1 floorplan is shown in Figure 2.15. The eight high-speed serial links[6] add multiple complexities to the physical design of Hurricane1, which are listed below.

- Each serial link requires a bias current. The chip receives a bias current externally which is mirrored in a current mirror circuit and distributed to each serial link instance. This routing requires special handling in the place-and-route tool to ensure it is shielded and sized properly. This can be avoided by generating bias currents on-die using current digital-to-analog converters (DACs), if the tolerance allows it.

- Each serial link macro corresponds to a memory channel that is driven by the L2 cache. Because of the available pin locations, the serial links must be placed a certain distance apart, which causes some to be further from the L2 cache than others. This results in a design that either has a long critical path determined by the furthest macro or an imbalance in the memory request latency if the design adds buffer registers asymmetrically.

- The L2 cache is placed and routed hierarchically in this design. However, this causes a large routing blockage to the serial links on the bottom edge of the die. This problem can be avoided by placing and routing the L2 cache at the top level of the design.

[6]The serial links are labeled SERDES in Figure 2.15.

2.3 Hurricane2: A RISC-V SoC with Dual-Lane Vector Unit and DVS

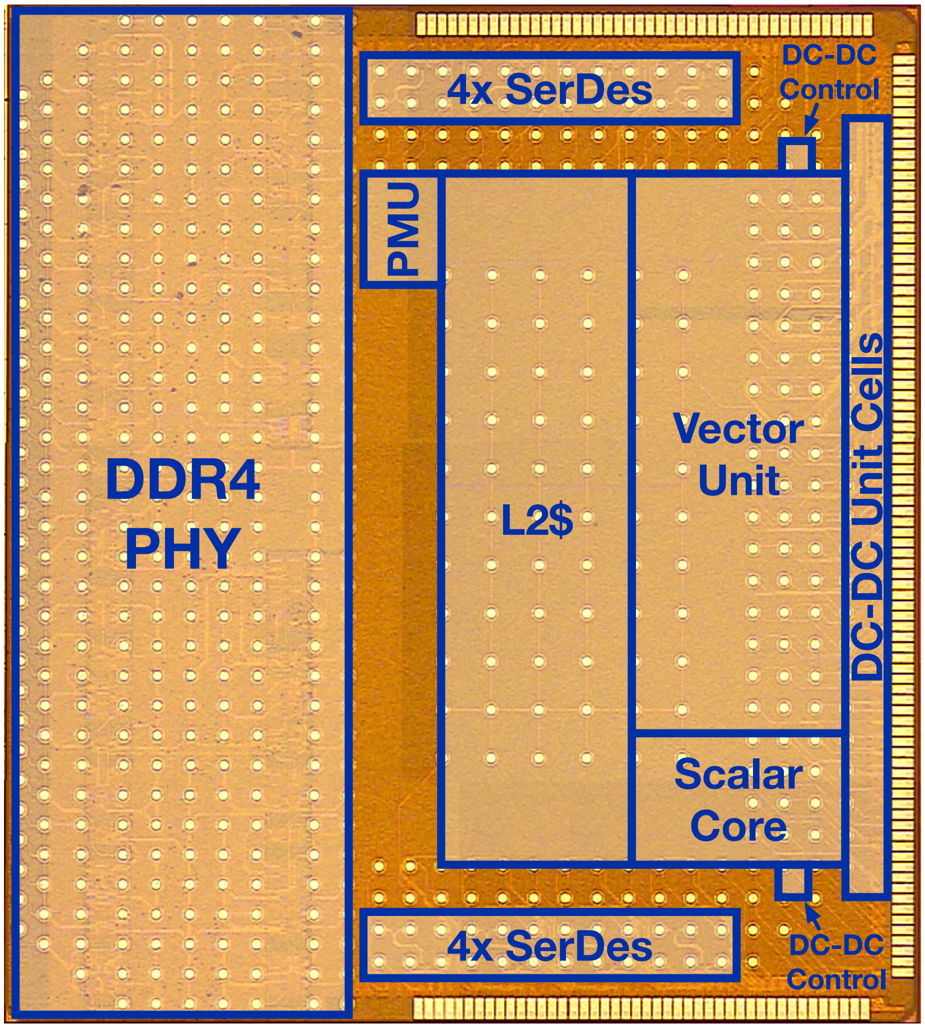


Figure 2.16: Annotated Hurricane2 die micrograph [81] (© 2020 IEEE).

2.3.1 Background

Hurricane1 demonstrated effective use of multi-core fine-grain dynamic voltage scaling, but lacked key components necessary to fully realize its potential. The Hurricane2 system-on-chip expands upon this technique with the addition of dedicated power management hardware to control the voltage states of the on-die voltage converters. In lieu of a second applications core, Hurricane2 includes a smaller RISC-V core which can actuate voltage state transitions in response to observed behavior on die. To facilitate this, Hurricane1 includes additional microarchitectural counters to indicate patterns in cache activity and power consumption [80].

Like Hurricane1, Hurricane2[7] [81] targets edge device applications. To maintain an equivalent vector compute throughput to Hurricane1 with a single applications core, Hurricane2 includes a dual-lane vector unit, which can perform the same maximum number of floating point operations per cycle as the two vector units on Hurricane1. Hurricane2 also includes a DDR physical layer (PHY) macro and controller to allow the test chip to have realistic DRAM latency when compared with a real productized SoC. Unlike Splash2 or Hurricane1, Hurricane2 was built using an early beta version of chisel3[8], which provided many additional features like multiple clock domain support. Native multi-clock support is an important feature when building modern digital systems, especially ones with multiple voltage-clock domains and high-speed off-chip I/O.

2.3.2 Architecture

Hurricane2 is a single-core, 64-bit RISC-V machine implementing the RV64G ISA[9] with the Xhwacha vector extension to program the included dual-lane vector accelerator. A die micrograph is shown in Figure 2.16. This core and the vector accelerator are powered independently by on-chip rippling switched-capacitor DC-DC converters [60] The clock for each rippling domain is generated using an adaptive clock generator [66]. A reduced-feature power management core is included to actuate voltage state transitions in these domains. A system block diagram is presented in Figure 2.17.

The scalar core contains separate 16 KiB L1 instruction and data caches and the system includes a unified 256 KiB L2 cache. The scalar core is capable of computing a single- or double-precision fused multiply-add (FMA) each cycle. A dual-lane instance of the Hwacha vector accelerator [70] is connected to the applications core using the Rocket custom co-processor interface (RoCC). Each vector lane contains a 16 KiB vector register file (VRF), a mixed-precision (half, single, double) FPU, integer support, a vector memory unit, and a 4 KiB vector instruction cache. The vector register file is organized into four banks of 4KiB SRAMs, and the vector memory unit has a 128-bit bus which can fully saturate the vector lane, allowing up to 16 bytes of data to be processed each cycle, corresponding to 4 double-precision, 8 single-precision, or 16 half-precision FMAs[10] per cycle.

[8]Benjamin Keller led the Hurricane2 tapeout. Others contributed the DC-DC converter design and vector accelerator design. The DDR PHY was donated by Cadence Design Systems. The author contributed the serial link transmitter design and transceiver back-end RTL; assisted with system-level RTL, integration, layout; and led testing for the manuscript.

[8]https://github.com/chipsalliance/chisel3

[9]Hurricane2 implements version 2.1 of the base ISA and version 1.9 of the privileged specification.

[10]One FMA is two floating point operations: multiply and add.

All cores and accelerator lanes share the 4-bank, 8-way 256 KiB L2 cache. The backside of the L2 cache can support one of three off-chip memory interfaces:

1. A low-speed, 4-bit parallel interface using standard I/O cells (LBWIF).
2. A high-bandwidth interface (HBWIF) [53] using 8 5 Gb/s serial links.
3. A DDR4 controller test site.

The configurable memory traffic switcher is able to select between these three interfaces at runtime, allowing experimental memory interface test sites to be disabled in the event that there is a functionality or power issue.

2.3.2.1 Power management

The power management unit (PMU) included on the Hurricane2 system-on-chip is an integer-only Rocket core with separate 4KiB instruction and data caches and no privilege modes. This core has access to the entire system memory map, allowing it to read counters measuring a variety of data and control the voltage states of the scalar applications core and vector unit. The L2 cache includes microarchitectural counters which count the number of L2 hits and misses and the number of outstanding transactions. Additionally, the system-on-chip includes an array of novel ring-oscillator-based temperature sensors, NORTHS [67], which provide spatial temperature information. The Hwacha vector unit also tracks the type and number of instructions pending, the count of instructions in flight, and the number of outstanding memory transactions, which allow the PMU to monitor the utilization of the vector lanes. Along with the standard architectural control and status registers (CSRs) mandated by the RISC-V specification, these registers provide visibility into all aspects of the system utilization.

2.3.3 Results

The chip test environment is identical to that used for Hurricane1 except with a Hurricane2-specific daughter card, which includes the packaged die and DRAM components. To determine the maximum operating frequency at various voltage levels, the DC-DC converters are bypassed while the voltage is held at a fixed DC value. The core and vector unit are clocked via an external clock generator while the clock frequency is increased until a self-checking vector DGEMM test program fails. This frequency is recorded for each voltage level and plotted in Figure 2.18.

The total energy per task is recorded by sampling current throughout a series of DGEMM loops to determine the total energy. The most energy efficient operating point is found to be 780 mV at 115 MHz. At this operating point the vector accelerator achieves 22.3 double-precision GFLOPS/W and runs 13× faster than on the scalar core alone. This is

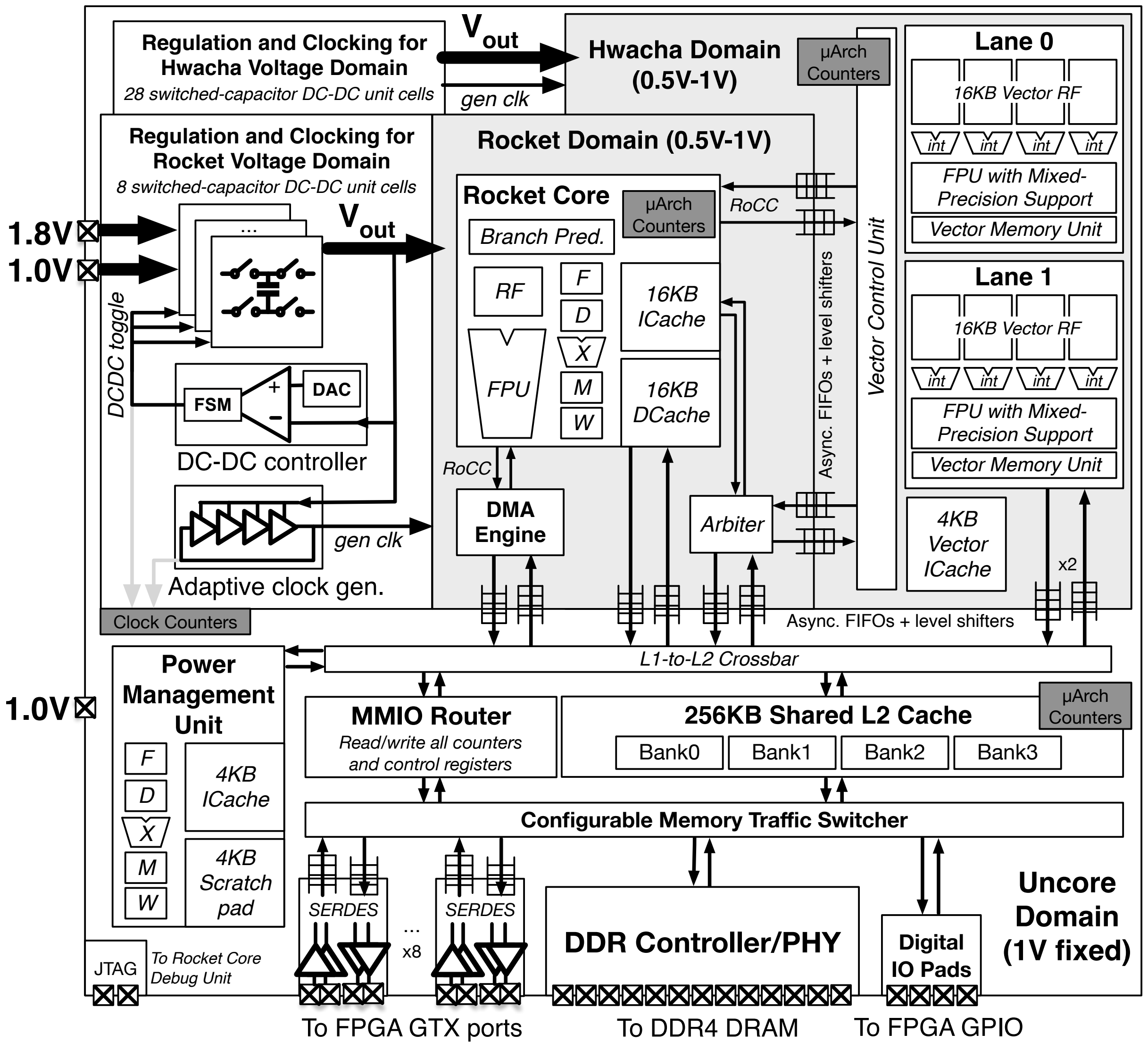


Figure 2.17: Hurricane2 block diagram [81] (© 2020 IEEE). Note that the previously published version of this diagram contains an error: The Rocket L1 caches are both 16 KiB.

a 12.6× energy efficiency improvement over the scalar core, which achieves 1.76 double-precision GFLOPS/W at this operating point[11]. [12] The vector unit also achieves 36.5 half-precision GFLOPS/W at this operating point, but the scalar unit does not support native half-precision arithmetic. A comparison of this system-on-chip with other state-of-the-art chips is shown in Table 2.3.

The efficacy of the power management approach is demonstrated by comparing three adaptive voltage scaling (AVS) algorithms, along with a control group with no AVS, across three synthetic benchmarks. Each benchmark alternates between computing a median filter and performing a general matrix multiply (GEMM) on either 24-, 64-, or 128-element square matrices. The results of this experiment are shown in Figure 2.19 The first, *Simple*, replicates the adaptive voltage scaling technique used in [65]. This algorithm observes the toggle rate of the DC-DC converters, which is proportional to average current draw, and increases the voltage mode when the toggle rate becomes more frequent than a programmable threshold. Conversely, it decreases the voltage state once the toggle rate decreases below a separate threshold, giving the control loop a programmable level of hysteresis.

The two other AVS algorithms, *AVS1* and *AVS2*, demonstrate the efficacy of the microarchitectural counters included on the Hurricane2 system-on-chip by monitoring the L1 and L2 cache miss rates, respectively. When the monitored cache miss rate increases above a programmable threshold, the power management code infers that the application core is in a memory-bound execution phase and lowers the core voltage. Similar logic is applied for low cache miss rates: Once the miss rates cross below a threshold, the application core is assumed to be in a compute-bound phase, and the core voltage is increased. Low numbers of cache misses can be caused either by heavily-compute bound phases or by idle loops. The RISC-V instructions retired (**instret**) CSR can be used to differentiate between an idle phase and a compute-bound phase.

The 24-element matrices fit entirely within the L1 cache, which quickly leads to no cache misses once the cache has warmed. Because of this, the *Simple* AVS algorithm performs worse than no algorithm at all, since there is no need to lower the core voltage due to cache misses. The *AVS1* and *AVS2* algorithms effectively do no voltage state changing once the cache has warmed, so it is unsurprising that they perform similarly to no AVS algorithm. The 64-element matrices fit in the L2 cache, but not the L1, which causes all L1 cache misses to hit in the L2 after warming, leading to mispredictions of low activity and causing the *AVS1* algorithm to perform worse than *AVS2*. As the 128-element matrices fit in neither cache, L1 misses translate more frequently to L2 misses, which causes the *AVS1* and *AVS2* algorithms to perform similarly. The *Simple* AVS algorithm only performed better than no AVS at all for matrices of this size.

[12]This calculation includes the leakage from the vector unit, which is unused during the scalar core measurement, so the improvement, while real, is artificially high.

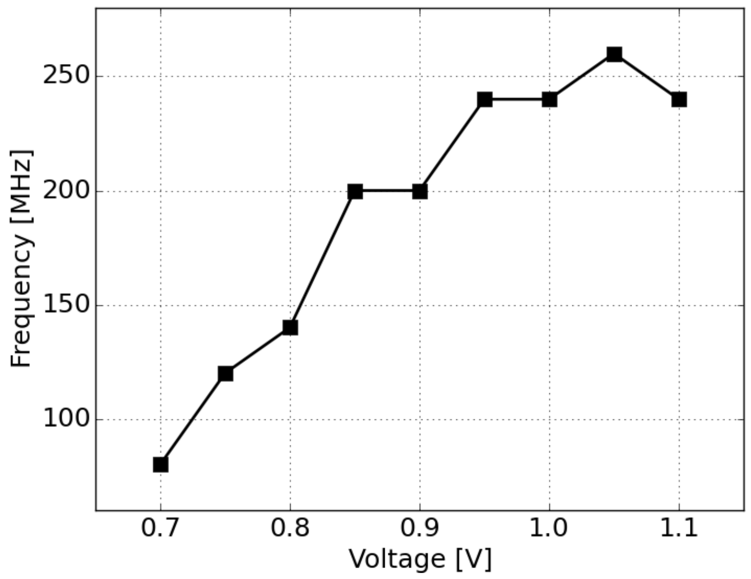


Figure 2.18: Maximum operating frequency versus core supply voltage on Hurricane2 [81] (© 2020 IEEE).

Table 2.3: Hurricane2 compared with prior art [81] (© 2020 IEEE).

Reference	Hurricane1 [57]	Hurricane2 [81]	JSSC '16 [65]	JSSC '15 [60]	APEC '14 [82]	ISSCC '17 [83]
Technology	28nm FD-SOI	28nm FD-SOI	28nm FD-SOI	28nm FD-SOI	22nm FinFET	28nm FD-SOI
Die size (mm^2)	7.84	16.7	3.03	2.37	160	1.87
Off-chip components	No	No	No	No	Package	N/A
Peak energy efficiency	19.6 GD-FLOPS/W	36.5 GH-FLOPS/W	41.8 GD-FLOPS/W	26.2 GD-FLOPS/W	Unspecified	10 TOPS/W †
DVS transition time (µs)	0.5	0.5	0.5	N/A	0.5	Unspecified
Voltage domain granularity	2.5 mm^2	0.5 mm^2	3.03 mm^2	2.37 mm^2	$<$ 0.5 mm^2	0.9 mm^2

† 1 four-bit MAC = 2 operations

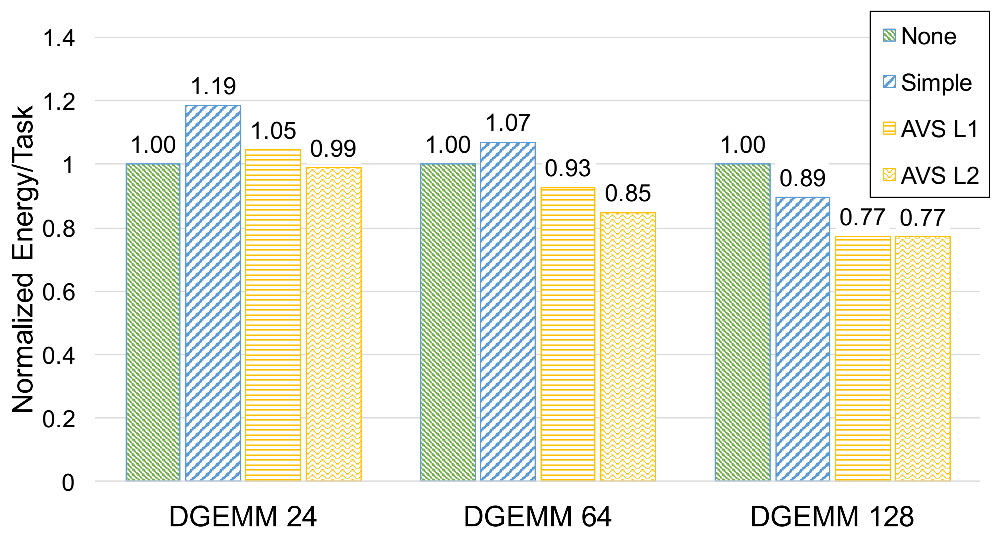


Figure 2.19: Comparison of Hurricane2 AVS algorithms [81] (© 2020 IEEE).

2.3.4 Physical design challenges

The Hurricane2 floorplan is shown in Figure 2.20. The Hurricane2 physical design flow fixes a number of issues present in the Hurricane1 flow. The L2 cache is no longer a hierarchical cell, fixing challenges with memory bus routing to the serial links. The serial link lanes are individually placed and routed as hierarchical cells that are integrated into the top-level design, allowing the floorplan of the individual lanes to be optimized separately from the top-level and reducing the complexity of the top-level place-and-route. The serial links are split into two dense banks with individual bias currents, reducing the amount of analog routing in the top level of the design and reducing the spatial distribution of memory bus wiring.

However, Hurricane2 is significantly larger than Hurricane1 at 16.77 mm^{2}and 7.84 mm^2, respectively, an increase of 113%. Much of this area is contributed by the DDR PHY, which is 35% of the total area of the Hurricane2 die and roughly half of the additional area relative to Hurricane1. The addition of the DDR PHY and controller adds significant complexity to the physical design flow, as it requires additional clock constraints, boundary I/O constraints, and power connections[13].

The L2 cache in Hurricane2 is 256 KiB, despite the floorplan being sized for a 512 KiB L2 cache, as can be seen in Figure 2.20. This is because the routing congestion between the cache and the compute modules (scalar core and vector unit) becomes significant enough to

[13]The DDR PHY power connection was a specific problem for Hurricane2: A misunderstanding with the integration of the third party DDR PHY led to a disconnected analog power supply within the PHY which rendered it inoperable. An attempt was made to use a focused ion beam (FIB) to repair a small number of parts to no avail. It is for this reason that all measurements in this document use the other memory interfaces.

prevent the routing tool from producing a correct design[14].

Hurricane2 is assembled by using a flip-chip process and a ball grid array (BGA) package[15], unlike Splash2 or Hurricane1, which use wirebond processes. In a wirebond process, a machine attaches small wires to bond pads on the chip and bond pads on either a package substrate or the printed circuit board (PCB) itself. For many academic chips, wirebonding directly to a PCB is the most logical option due to its low cost. However, bond wires have significant inductance which limit data rates and cause supply voltage to drop (or spike) when there is a large increase (or decrease) in activity due to Faraday's Law. Wirebonding also requires the chip I/Os to be placed along the perimeter of the die. For large dies this has two consequences:

1. The center of the chip is far away from the power connections, increasing the effective power grid resistance and causing an unwanted voltage drop gradient.
2. The number of I/Os becomes severely limited relative to the logic complexity. This is demonstrated in equation 2.4, which indicates that the I/O count is proportional to the die perimeter. By comparison, the I/O count of a flip-chip circuit is proportional to the area, as demonstrated in equation 2.5.

$$N_{IO,wirebond} \propto W_{die} + H_{die} \tag{2.4}$$

$$N_{IO,flip-chip} \propto W_{die} \times H_{die} \tag{2.5}$$

In a flip-chip assembly process, the die is manufactured with small solder bumps across the entire surface. The die is "Flipped" solder-side down onto a package and heated to melt the solder. The package has a larger pin pitch that is more suitable for PCB assembly than the fine solder ball pitches on the die. By using a lower cost fabrication process than the die itself for the package, this technique compromises I/O density with cost. The large number of I/Os in the DDR PHY required the use of flip-chip on Hurricane2.

[14]Routing tools will abort after a certain number of iterations, leaving a design with open or short circuits.

[15]It was unknown at the time of tapeout if Hurricane2 would be packaged. To de-risk the project, a subset of I/Os were also connected to wirebond pads, which can be seen in the die micrograph in Figure 2.16.

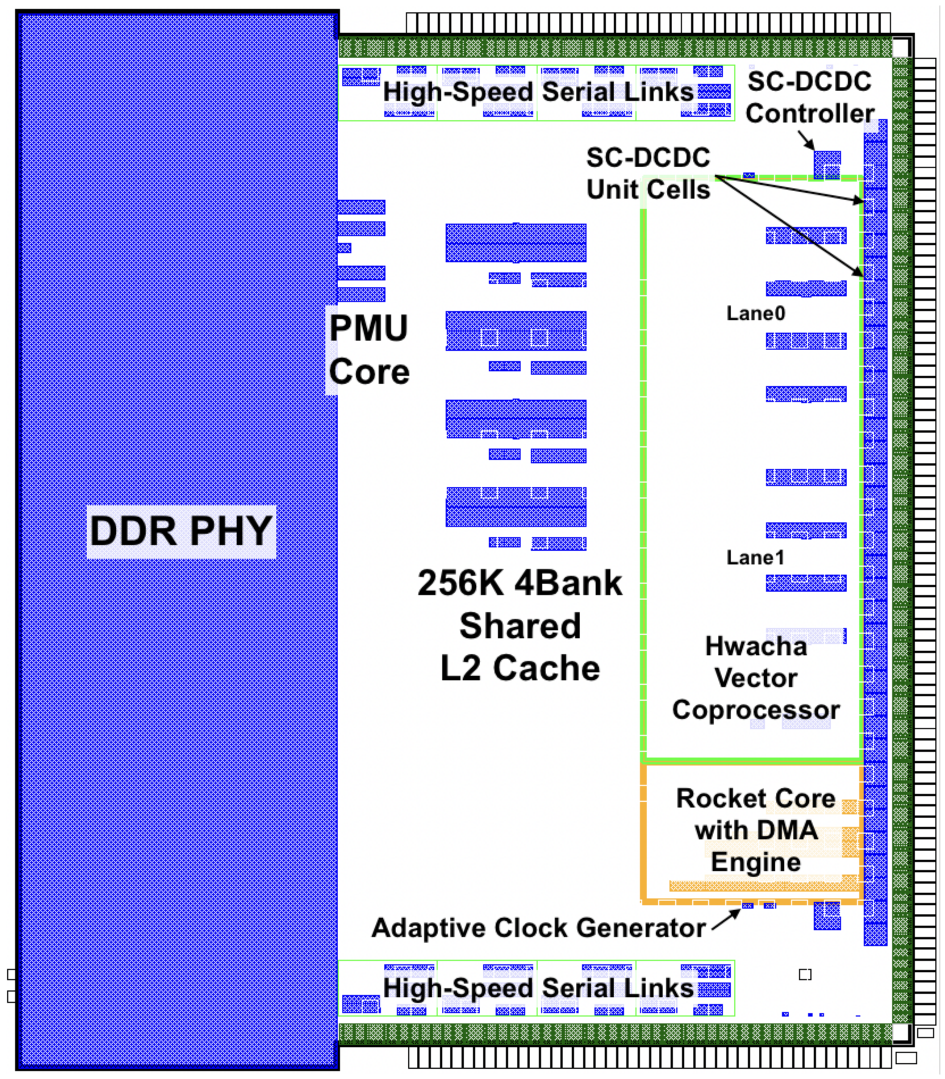


Figure 2.20: Annotated Hurricane2 floorplan [80] (© 2017 Benjamin Keller).

Chapter 3

Generated Multicore Systems-on-Chip in 16nm FinFET

The FinFET is a three-dimensional transistor developed to solve problems with the performance and manufacturing of SOI devices [84]. FinFET technologies saw commercial use starting at the 16nm node and, at the time of this writing, are planned for nodes through 3nm [85]. The FinFET transistor provides significantly more control over the channel due to its geometry, leading to lower leakage than a planar technology with an equivalent gate capacitance. Systems-on-chip manufactured in FinFET technologies are ubiquitous in products today.

3.1 Eagle: An 8-core Generated RISC-V SoC

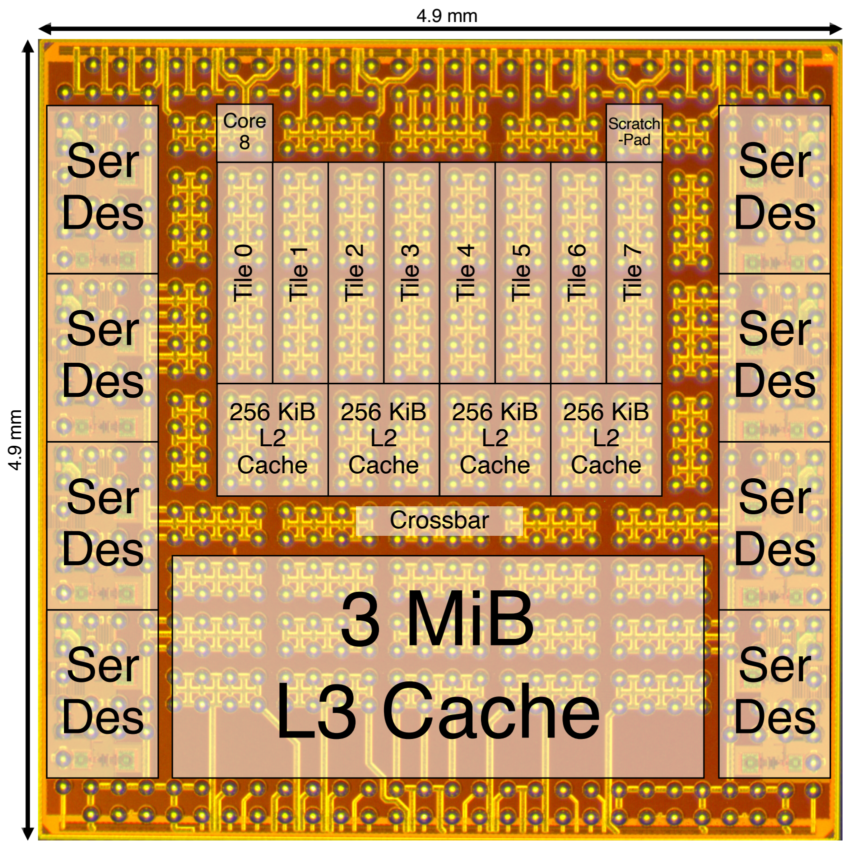


Figure 3.1: Annotated Eagle die micrograph [86] (© 2021 IEEE).

3.1.1 Background

Prior Chisel-based test chip designs demonstrate the efficacy of a generator-based approach to building systems-on-chip, but these designs lack the size and complexity of productized edge systems-on-chip needed to run modern workloads like deep neural networks (DNNs). Eagle[1] [86], shown in Figure 3.1, is a multicore edge system-on-chip generated using Chisel and the Berkeley analog generator (BAG) to prove the scalability of this methodology. In addition, Eagle includes many system features absent in the prior academic test chips presented in this work, including a boot ROM, standard I/O peripherals like SPI and I^{2}C, JTAG, and 3 levels of cache hierarchy.

[1]Eagle was co-led by the author and Colin Schmidt. Zhongkai Wang led the serial link development with the help of others. SiFive donated the multi-level cache generator.

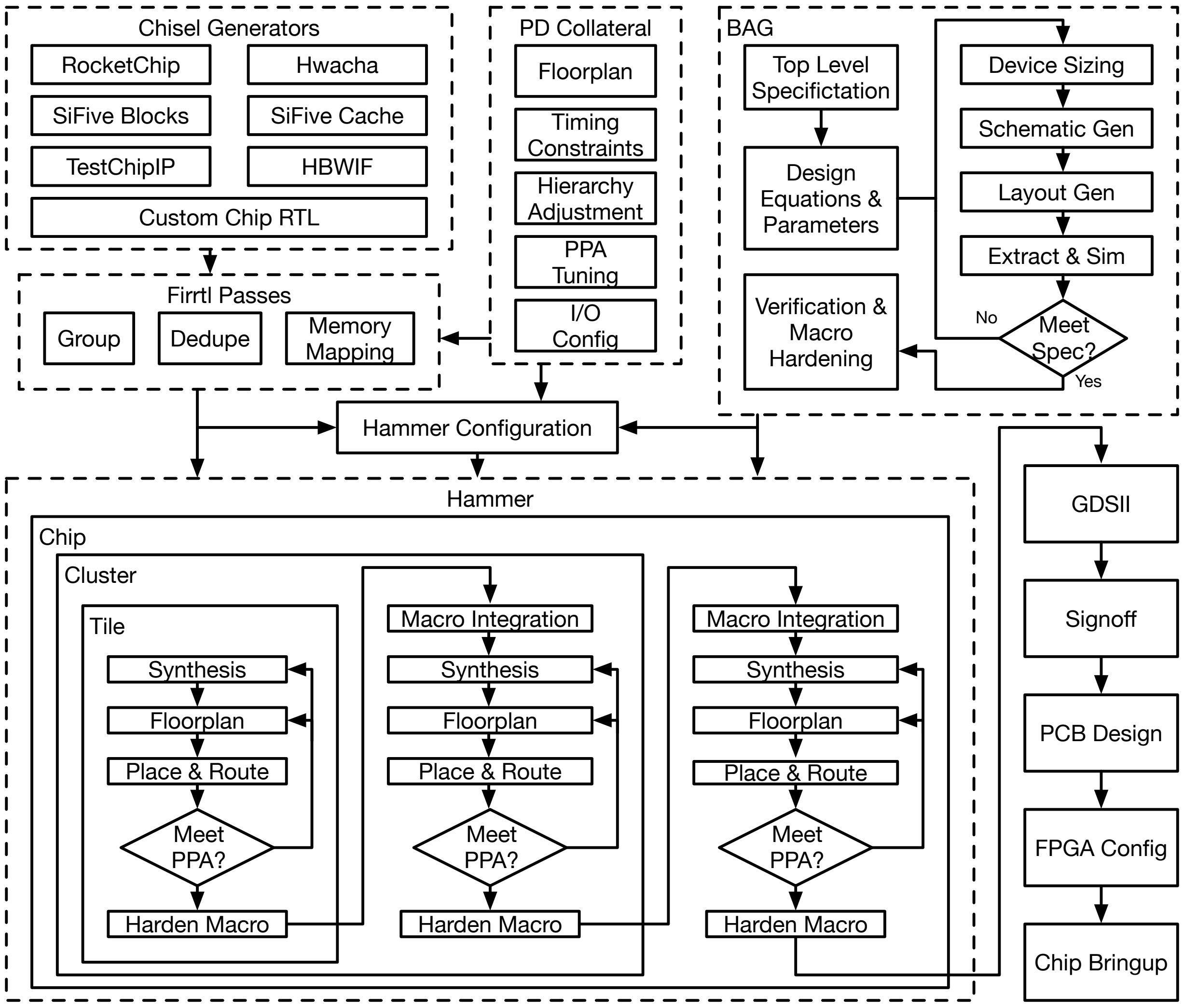


Figure 3.2: Eagle chip generation flow.

Eagle is developed using a novel physical design framework, Hammer [17], which is described in greater detail in Chapter 4. This design flow is presented in Figure 3.2. A key contribution is the automated approach to hierarchical design, which augments commercial EDA tools with a technique to modify the design hierarchy and create build rules for streamlining the implementation and integration of hierarchical cells within the flow.

3.1.2 Architecture

[2] The Eagle system-on-chip is manufactured in TSMC 16nm finFET and measures 4.9 mm by 4.9 mm, as shown in Figure 3.1. Eagle contains eight application cores, one system

[2]This section contains excerpts that are © 2021 IEEE [86].

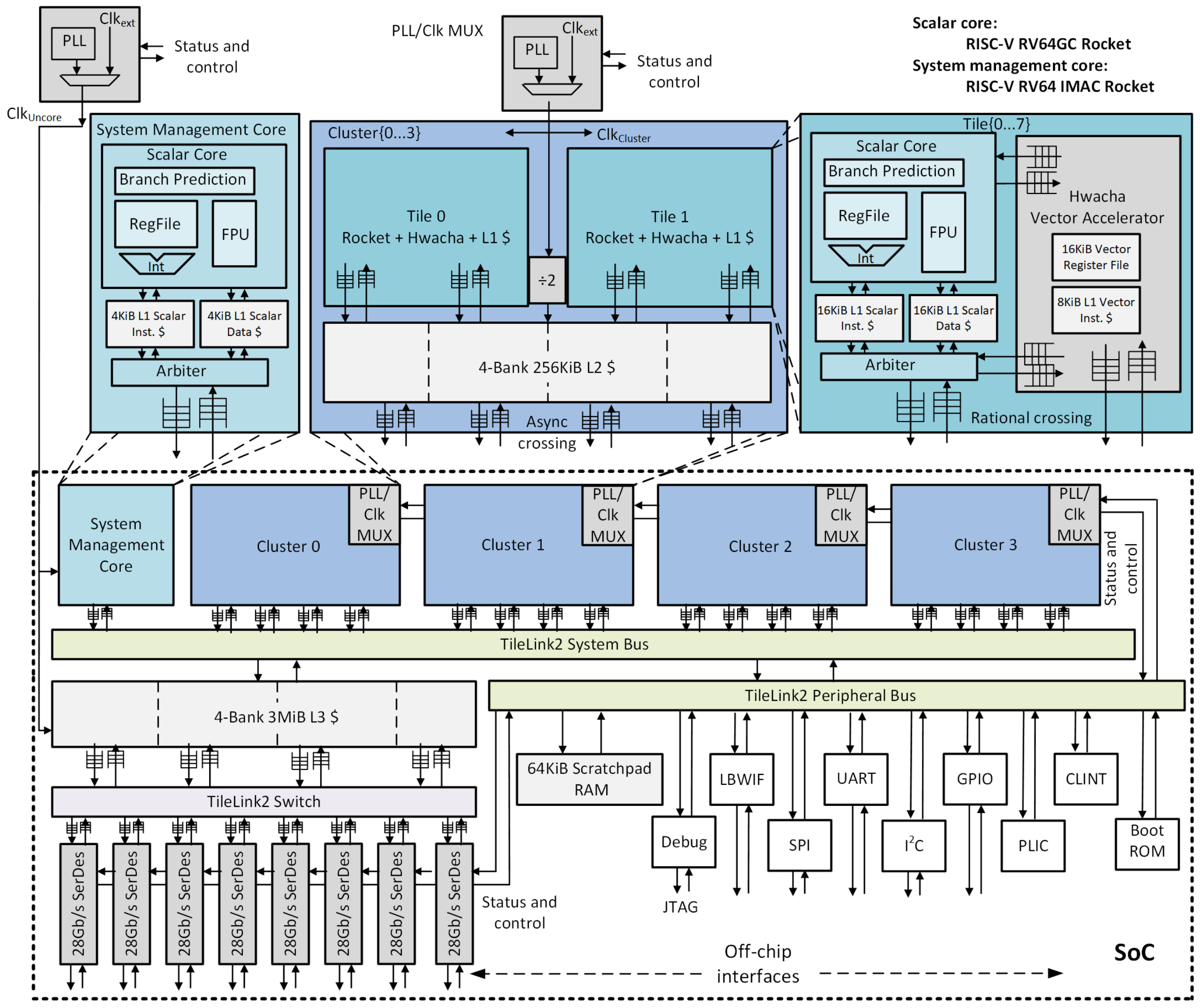


Figure 3.3: Eagle block diagram [86] (© 2021 IEEE).

management core, eight serial links, and three levels of cache, as shown in Figure 3.3. The application cores comprise one scalar RISC-V processor and a decoupled vector accelerator. The application cores measure 332 mm by 1658 mm with a routed gate density of 64.1%. The scalar processor is a single-issue, in-order, 5-stage processor implementing the RV64GC ISA and is capable of up to 2 floating point operations per cycle. The cores additionally support a non-standard Hwacha extension to interface with the vector accelerator, which is shown in Figure 3.4. This vector accelerator supports double-, single-, and half-precision floating point operations, with a maximum of 8, 16, and 32 operations per cycle, respectively, per lane. Such variable precision is useful for both inference and training [87]. Each lane uses a banked memory structure with bank-local compute units for simple arithmetic, and longer latency functional units shared amongst banks as shown in Figure 3.4. The lanes are computationally balanced such that a fully saturated memory interface can match the

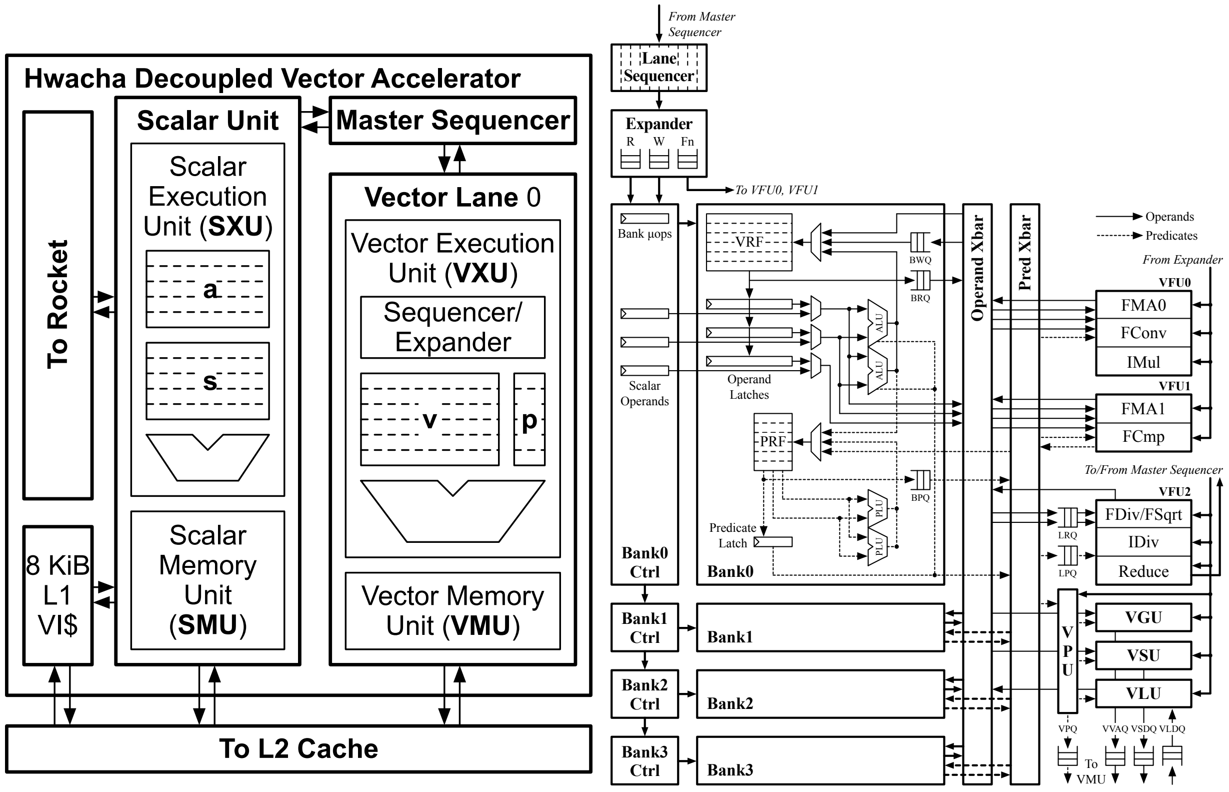


Figure 3.4: Block diagram of the Hwacha vector unit on Eagle [86] (© 2021 IEEE).

arithmetic throughput of two ongoing FMAs that each use a single scalar input and two vector inputs, as would be found in the inner loop of a matrix multiplication kernel. The different precisions supported by the programmable-precision vector unit can be intermixed freely in code and fully interoperate. Each register in the machine has an assigned precision that determines the functional units used, how memory accesses should be performed, and what conversions, if any, should be applied. The registers can be reassigned a precision by the programmer at any point, and the vector unit will repack the register file to preserve efficient access patterns under the new assignment. The overall throughput of the vector unit is determined by the precision of its operand registers, with the machine supporting multiple precisions simultaneously at the cycle-level.

The application cores have a 16 KiB L1 instruction cache, a 16 KiB L1 data cache, and an 8 KiB L1 vector instruction cache. These L1 caches and the vector memory unit are backed by an inclusive, unified 256 KiB L2 cache that is shared between two cores, which collectively form a cluster. Within a cluster, the L2 cache runs at half the core frequency and communicates synchronously with the cores. Each cluster operates in its own individual voltage and frequency domain, which is asynchronous with the rest of the chip. There are a total of four clusters, and all are backed by an inclusive, unified 3 MiB L3 cache. The system also contains a ninth core, the system management core, which is a different generator

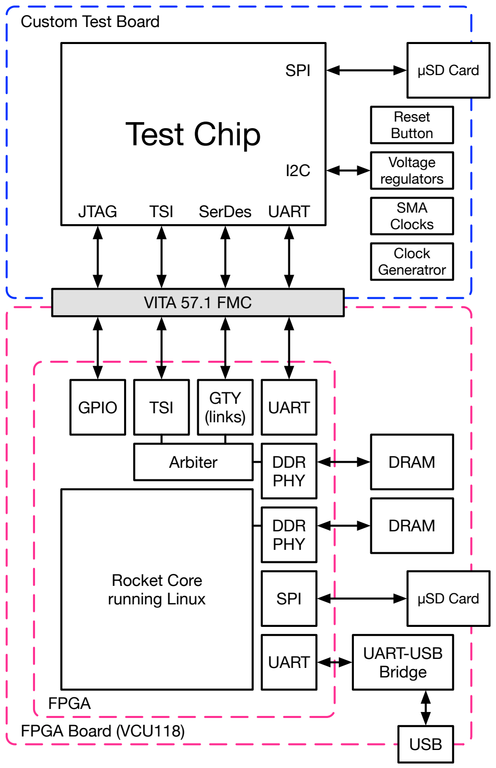


Figure 3.5: Eagle test setup and block diagram [86] (© 2021 IEEE).

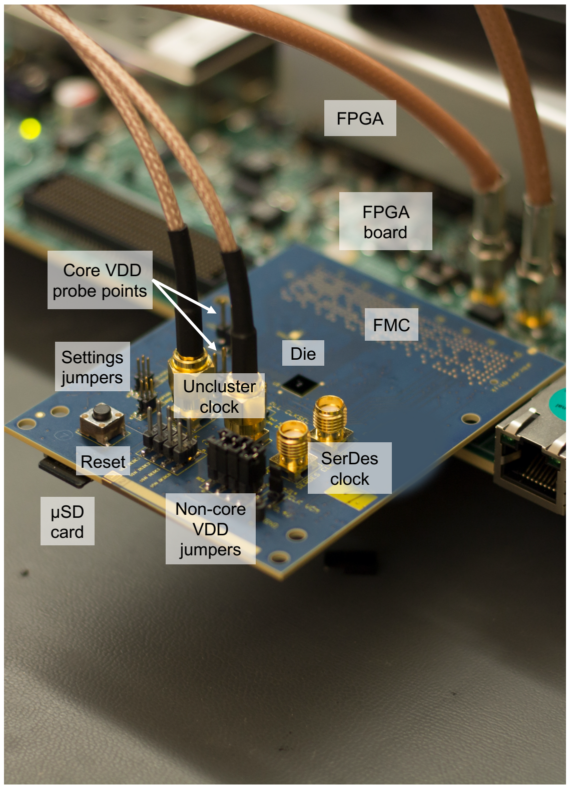


Figure 3.6: Eagle test board on lab bench with labeled components [86] (© 2021 IEEE).

configuration that supports a less complex ISA, RV64IMAC, has only 4KiB L1 caches, no L2 cache, no vector accelerator, and lacks user mode. Its L1 caches are backed directly by the L3 cache. The system management core has access to all the system control registers and memory space and is responsible for bringing the application cores out of reset and controlling on-board regulators over I2C and GPIOs. All of the cores are debuggable over JTAG. An on-board scratchpad memory is used by the system management core to load a bootloader from a microSD card over the SPI interface.

The L3 accesses main memory over the 8 serial links or a low-speed 4-bit-wide backup interface. The selection of a main memory interface is made at runtime through memory-mapped registers. Both of these interfaces communicate with a host FPGA board, a Xilinx VCU118, through a custom VITA57.1-compatible PCB with an FMC connector. The FPGA is programmed with logic that transforms the custom memory protocols used by these interfaces into memory requests to the host FPGA's DRAM controllers. The FPGA contains additional dedicated logic to access the chip's debug interfaces, including UART, JTAG, and GPIOs, which is controlled by a generated soft RISC-V core running Linux on the host FPGA.

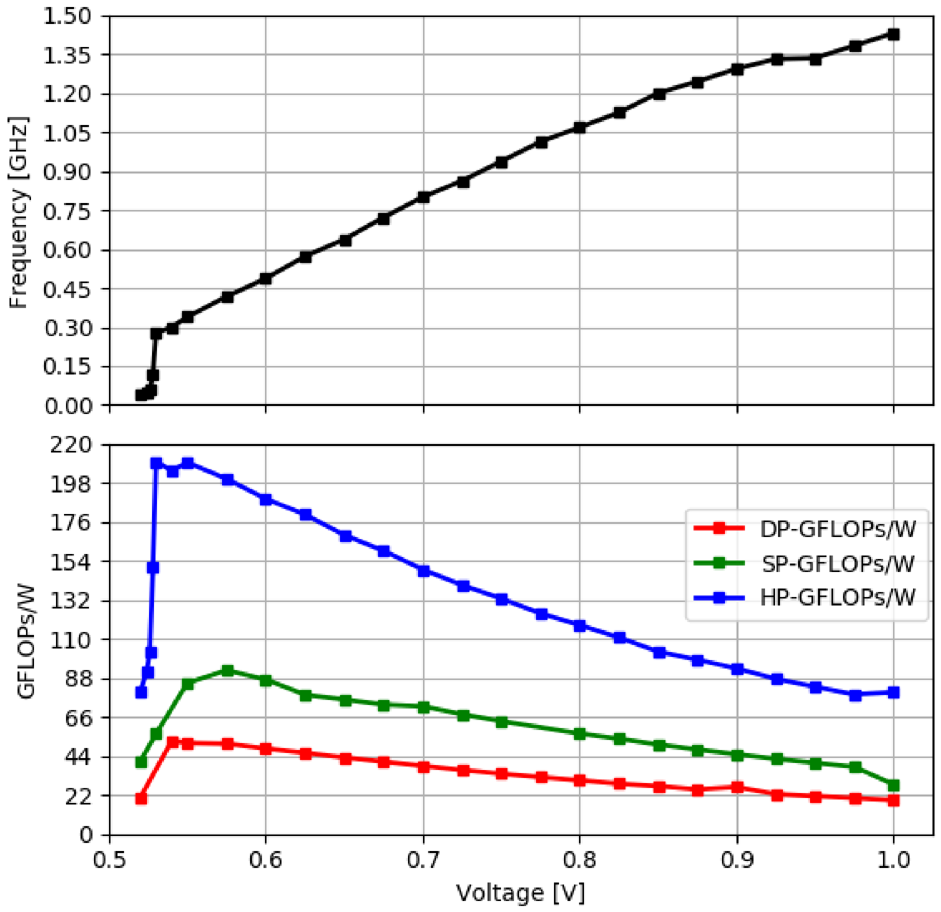


Figure 3.7: Eagle maximum frequency and GEMM energy efficiency [86] (© 2021 IEEE).

3.1.3 Results

The development of Eagle took 3 full-time and 4 part-time students approximately 7 months from concept to tapeout using an agile design methodology as shown in Figure 3.2. The SoC is a generated instance from a chip generator written in Chisel leveraging many pre-existing open-source Chisel RTL generator components in addition to new RTL generator components developed for this project. The clock generators, PLL supply regulators, and SerDes TX and RX blocks were generated using BAG. The Chisel generators use FIRRTL to produce synthesizable RTL and other ASIC design collateral, including memory map files, technology SRAM wrappers, and a Linux device tree string. Several additional FIRRTL passes are included to perform the technology SRAM mapping and to reorganize the logical hierarchy of the chip into the desired physical hierarchy, which is desirable when reusing existing RTL and for separating the logical definition of a circuit from underlying physical implementations. The physical design and physical verification flow is built using Hammer [17] on top of commercial synthesis and place-and-route tools. Hammer consumes physical design inputs such as floorplan files, timing constraints, and flow configurations in high-level, tool-agnostic

Table 3.1: Eagle compared with prior art [86] (© 2021 IEEE).

Ref.	Eagle [86]	ESSCIRC '16 [65]	VLSI '19 [88]	VLSI '19 [89]
Technology	16nm FinFET	28nm FD-SOI	16nm FinFET	16nm FinFET
Die size (mm^2)	24.01	3.03	25	15.25
Total Cores	9	2	3	496
ISA	RV64IMAFD Xhwacha4	RV64IMAFD Xhwacha3	Armv8-A	RV32IM
Total SRAM	4.5 MiB	80 KiB	9 MiB	2MiB
# Transistors	125M gates	568K std cells	>0.5 B	385M
Theoretical Peak Perf.†	368.4 GH-FLOPS	3.188 GD-FLOPS	16 GS-FLOPS	695 GRVIS
Peak energy efficiency†	209.5 GH-FLOPS/W	41.8 GD-FLOPS/W‡	58.7 GOPS/W	93.04 GRVIS/W
Floating-point Precisions Supported	Half, Single, Double	Single, Double	Half, Single, Double	None
Max Clock Frequency	1.44GHz	797MHz	>1 GHz	1.4 GHz

†Software programmable

‡No variable precision; GH-FLOPS/W will be similar

1 GRVIS = 1 Giga (10^9) RISC-V instructions per second.

data structures and uses APIs to generate scripts and other collateral needed to run a full physical design flow. The high-level APIs allowed different generator designs to be built quickly and evaluated for their power, performance, and area characteristics. Hammer provides mechanisms to give the user full control over the generated scripts by inserting custom command sequences before or after flow API steps or by replacing the steps entirely. As the generator design matures, this enables fine-tuned physical design optimizations to be put into place to produce an equally optimal design as a traditional, non-Hammer flow. Because the entire flow supports generators, smaller versions of the SoC were built and tested early in the design process, which allowed all team members to work on their contributions simultaneously. This enabled many iterations of each component to be elaborated, tested, and pushed through physical design and allowed collection of partial results to drive flow enhancements after each run.

A block diagram of the test setup is shown in Figure 3.5, and a photo of the test board with annotated components is shown in Figure 3.6. The maximum measured operating frequency of the cores is 1.44 GHz at 1.00 V. Figure 3.7 illustrates the maximum operating frequency and energy efficiency across the range of functional supply volatges. Per-cluster energy-efficiency is measured by running a dual-core matrix multiplication (GEMM) benchmark in a loop which transmits cycle and iteration counts over UART, while measuring cluster supply current. The cluster supply voltage and frequency are swept until failure while the uncluster (L3 and peripherals) is kept at 0.8 V and 100 MHz. GEMM is chosen as a benchmark because static timing analysis shows that the core critical path is through the vector floating point unit. In this design, the vector data bandwidth is limited to 50% of theoretical peak by L2 cache frequency division. A wider vector data L2 port would correct this, doubling the energy efficiency over these results for long vectors. At the most efficient operating point, 339 MHz at 0.55 V, the chip runs double-precision matrix multiplication (DGEMM) at 56.5 GFLOPS/W, single-precision (SGEMM) at 92.3 GFLOPS/W, and half-precision (HGEMM) at 209.5 GFLOPS/W, which correspond to 34.4%, 34.5%, and 33.0% of theoretical peak vector performance, respectively. This beats the state of the art, as shown in the comparison in Table 3.1.

3.1.4 Physical design challenges

3.1.4.1 Floorplan

While Hurricane1 and Hurricane2 were small enough to not strictly require hierarchical physical design, Eagle is sufficiently large at 125 million gates and 24.01 mm^{2}in 16nm FinFET that hierarchical design is mandatory to implement the chip. Eagle's chip-level floorplan is shown in Figure 3.8. As shown in Figure 3.2, Eagle uses a multi-level hierarchical design flow. The hierarchical relationships are illustrated in the diagram in Figure 3.9. The top-level contains 4 cluster cells, 8 serial link lane cells, 1 system controller cell, and 1 boot ROM cell[3]. Eagle uses a custom FIRRTL transform, **GroupAndDedup**, to modify the design hierarchy during FIRRTL compilation. This is very useful as it allows the designer to continue to write code with logical hierarchies that are intuitive, while keeping the generated hierarchy that makes sense for physical design. The cluster, tile, and serial link lane modules all make use of this feature.

The cluster and tile floorplans are shown in Figure 3.10. The serial link floorplan is shown in Figure 3.11.a, alongside a GDSII[4] in Figure 3.11.b), which shows in greater detail the power grid and inductor layouts in the serial link transmitter and receiver. The serial links are placed along the left and right edges of the chip to allow the differential signal traces to escape without requiring a vias on the board, which can add transmission line stubs and cause unwanted reflections. Using only the left and right edges and excluding the top and bottom allows a single layout of the serial link lane, which is mirrored on the right side. This

[3]While the boot ROM is small, making it a hierarchical cell allows it to be replaced without re-implementing the entirety of the top-level design. This is useful if the boot ROM code is under development and needs to be replaced. This accomplishes a similar goal to that of a via-programmable ROM, except that it cannot be replaced after tapeout. A via-programmable ROM would be a superior choice for this purpose, but one was not available at the time of Eagle's construction.

[4]GDSII is the file format for mask data

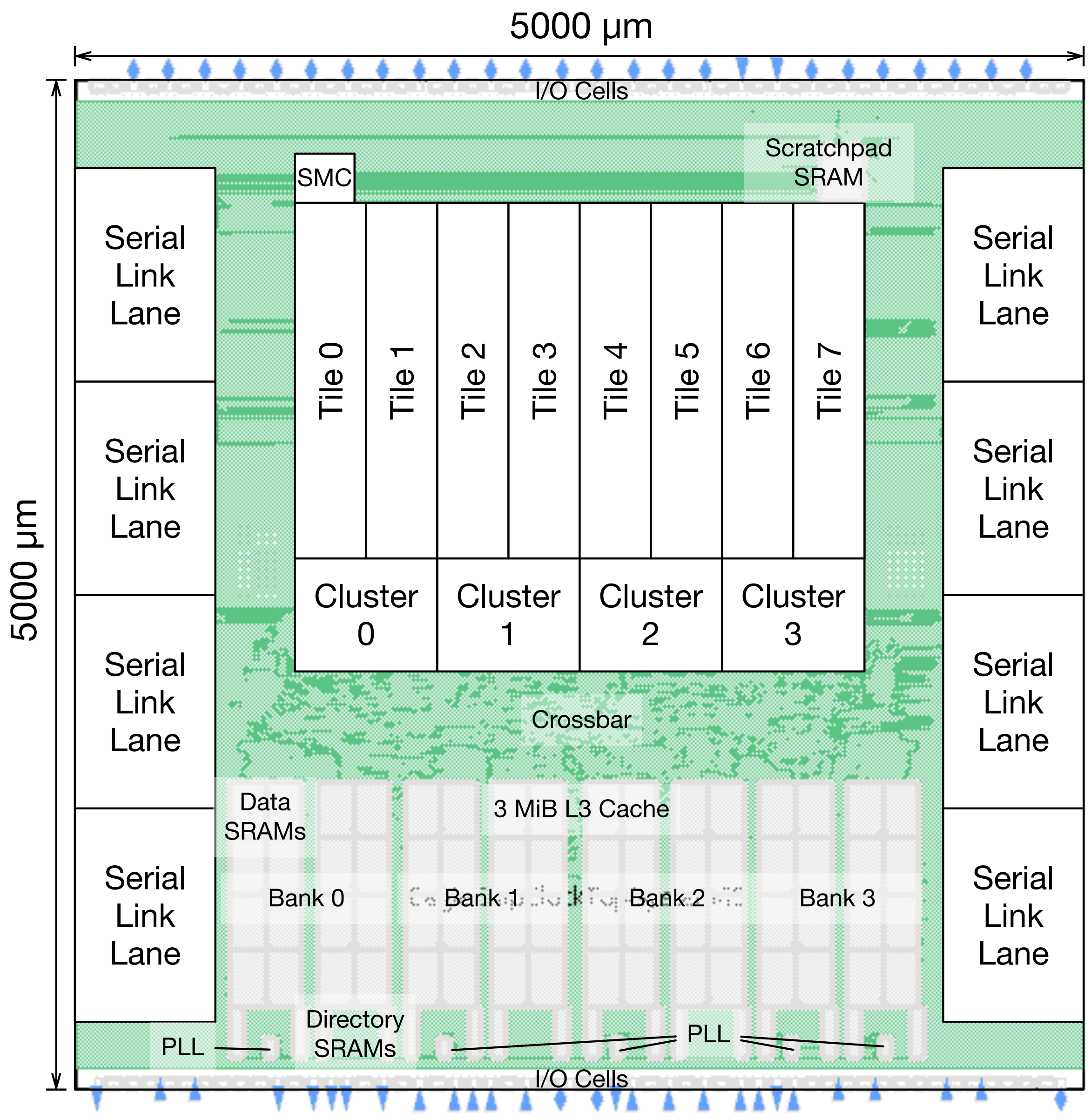


Figure 3.8: Annotated Eagle chip floorplan. Dimensions shown are as drawn before a 98% optical shrink.

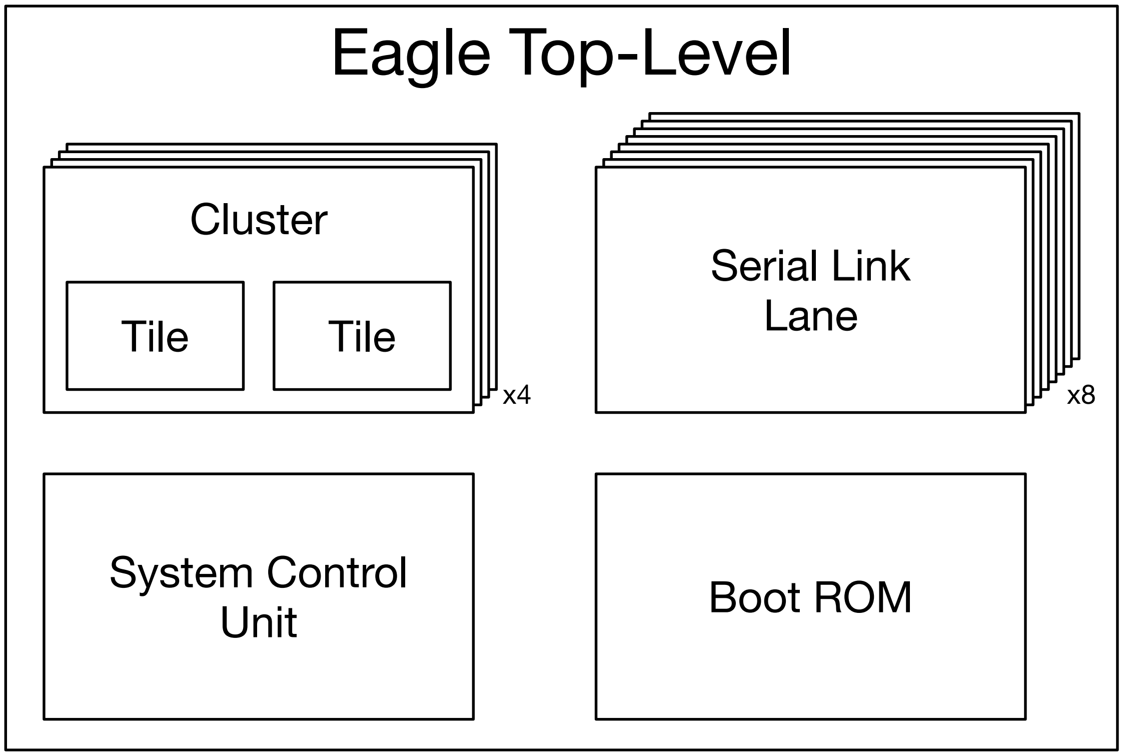


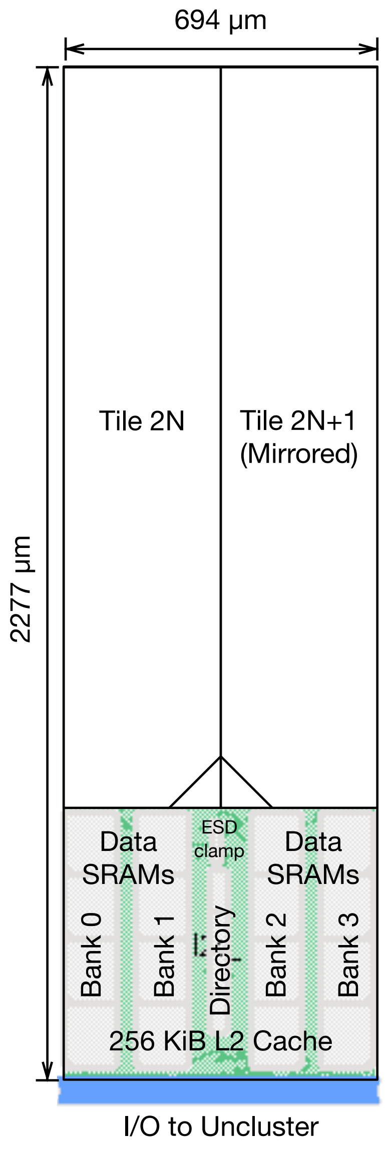
Figure 3.9: Physical design hierarchy of Eagle.

is important because FinFET processes do not allow ninety degree rotations of cells due to its unidirectional polysilicon requirement, unlike many planar technologies, including 28nm FD-SOI[5].

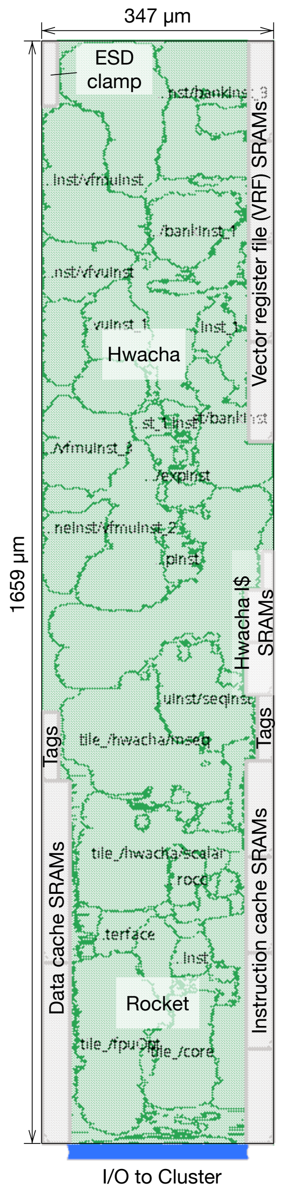
The Eagle die is assembled onto the PCB by soldering the bare die directly to a compatible footprint without a package[6], called chip-on-board (COB). Figure 3.12 shows the PCB footprint compatible with Eagle. The purple circles represent via locations on the board; this is a variant of via-in-pad, which places vias underneath a pad. Because of the size of the via's annular ring (9 mil or 228.6 µm) and the minimum spacing requirement in the chosen PCB manufacturing process, a single via must connect to multiple bumps on the die, which are 80 µmin diameter at a pitch of 170.05 µm. To facilitate this, the Eagle bump assignments are chosen with all interior bumps ganged into 2 by 4 groups, which are all power and ground signals. Bumps are removed above the inductors to improve their Q factor, and a region of bumps is removed at the bottom of the die to make room for five additional board vias, which are used for signal routing instead of power. All other signal routes are placed on the periphery of the bump-out to avoid the cost of fine-pitch PCB traces between bumps.

[5]In the Hurricane1 die micrograph, the rotated serial links are visible on the bottom edge of the chip.

[6]This assembly process was chosen to reduce cost.

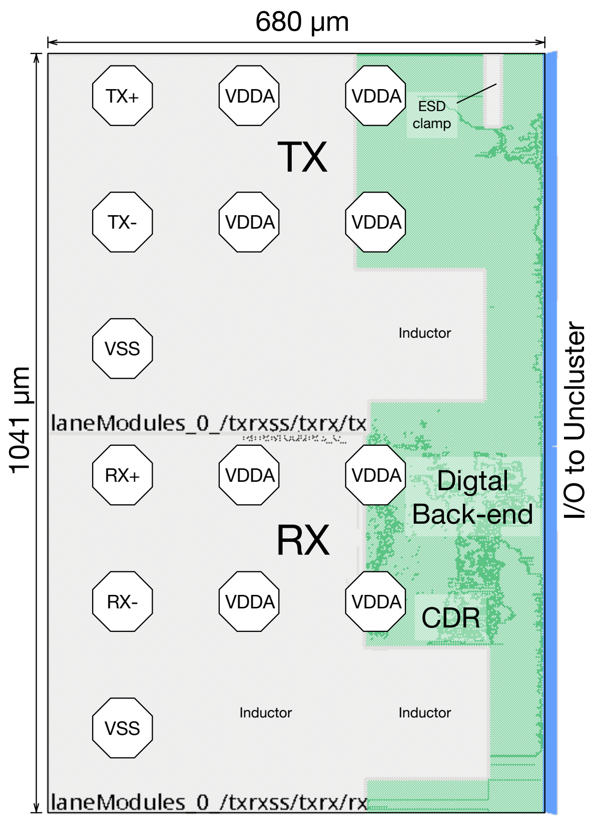


(a) Cluster

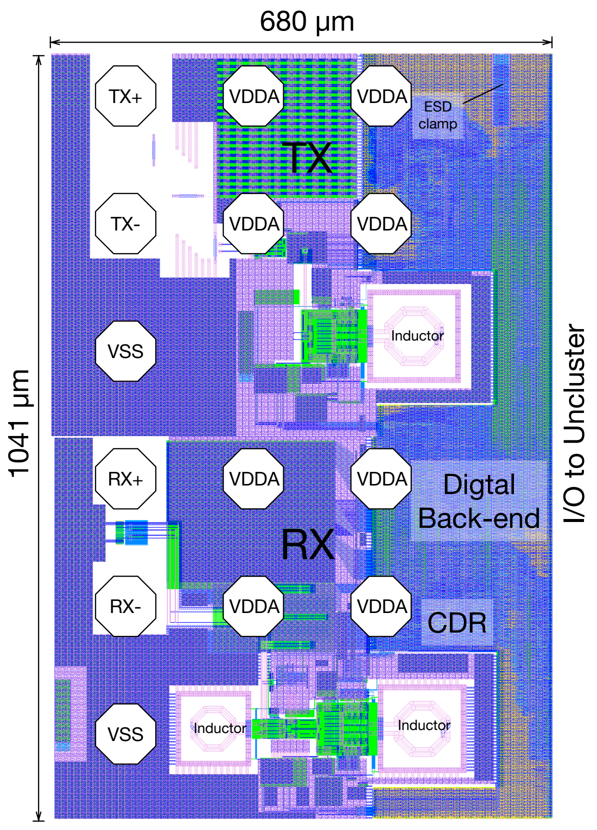


(b) Tile

Figure 3.10: Eagle cluster and tile floorplans. Dimensions shown are as drawn before a 98% optical shrink.



(a) Floorplan



(b) Layout

Figure 3.11: Eagle serial link lane floorplan and layout. Dimensions shown are as drawn before a 98% optical shrink. For clarity, only the upper metal layers are shown in the layout.

Differential signals connected to the serial links and clock receivers are shielded on either side with ground bumps (GSSG).

The groups of 8 power or ground bumps tie directly into the power grid below. The power grid strategy uses the top two layers for a dense power grid with sparser grids on lower layers. The top level power grids in the design connect by abutment, which provides superior utilization of routing layers and lower parasitic resistance than ring-based approaches. Because the entire chip shares a ground, all ground straps on the top two metal layers are directly connected. Supply voltage straps occupy the same tracks, but are left with a gap when transitioning between rails. In areas where two supplies are needed, the power rails alternate between the two supplies. Electrostatic discharge (ESD) clamps are placed throughout the design to prevent damage to the circuitry and comply with DRC rules.

Because each cluster is in its own voltage domain, the cluster supplies are present only above their respective cluster, while all share a common ground. This creates a requirement that the cluster widths be a multiple of the bump pitch for the physical voltage domains

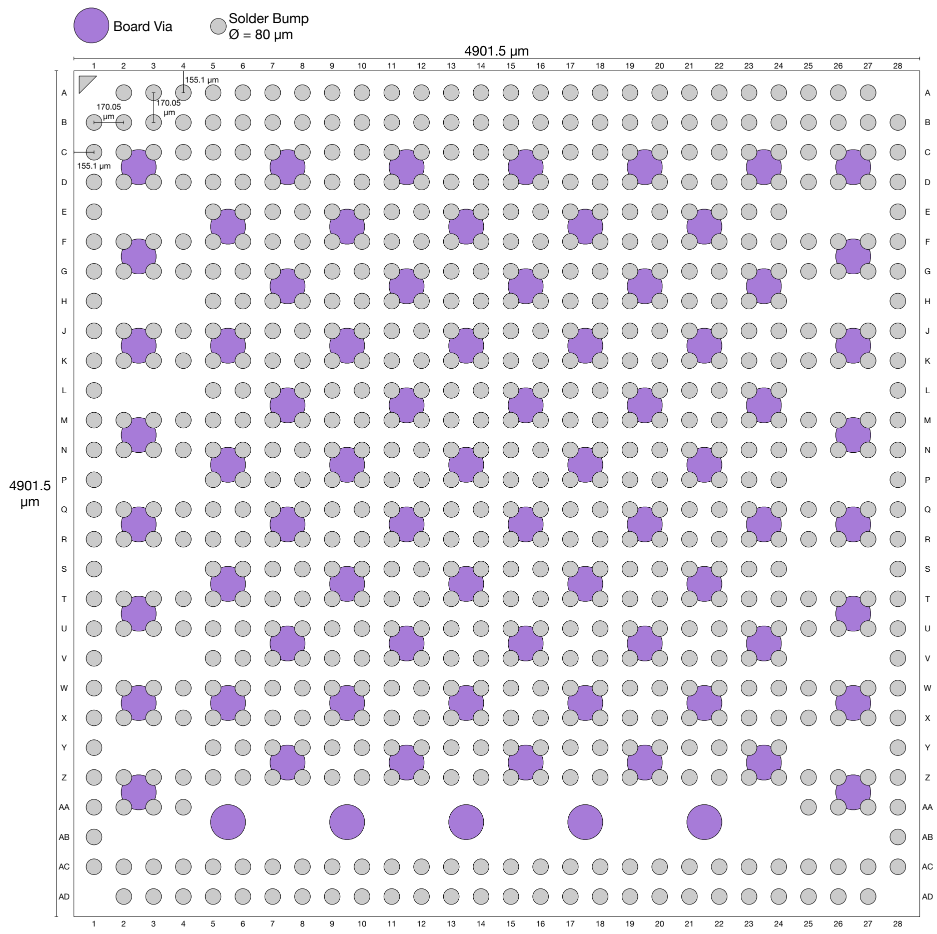


Figure 3.12: Eagle PCB footprint.

to line up with the bumps above, which contributes to the shape of the cluster and tiles. This indirectly ties the power strap X dimension pitch to the bump pitch, because the power strap pitch must evenly divide the tile width for multiple tiles to align with the top level grid when abutted. A similar constraint is imposed on the Y dimension pitch by the serial link lanes. An added complexity with the serial link lanes is that the BAG-generated power grid must match in order for the grounds to connect (the serial links have separate voltage rails). BAG requires all geometries to be on a common grid, usually the fin grid, further constraining the pitch. These series of power strap requirements are conceptually simple, but difficult to perfect in practice, making automation of power strap pitches and offsets a very useful feature of Hammer.

As with Splash2, Hurricane1, and Hurricane2, SRAM placement is a critical floorplanning task for Eagle, which is also made difficult by the necessary hierarchy manipulation in addition to the known challenges with floorplanning for generator-based designs. Figure 3.8 shows the SRAMs placed in the 3 MiB L3 cache, while Figure 3.10.a) shows the SRAMs placed in the 256 KiB L2 cache, and Figure 3.10.b) shows the Hwacha and Rocket L1 and

VRF SRAMs.

3.1.4.2 Clocking

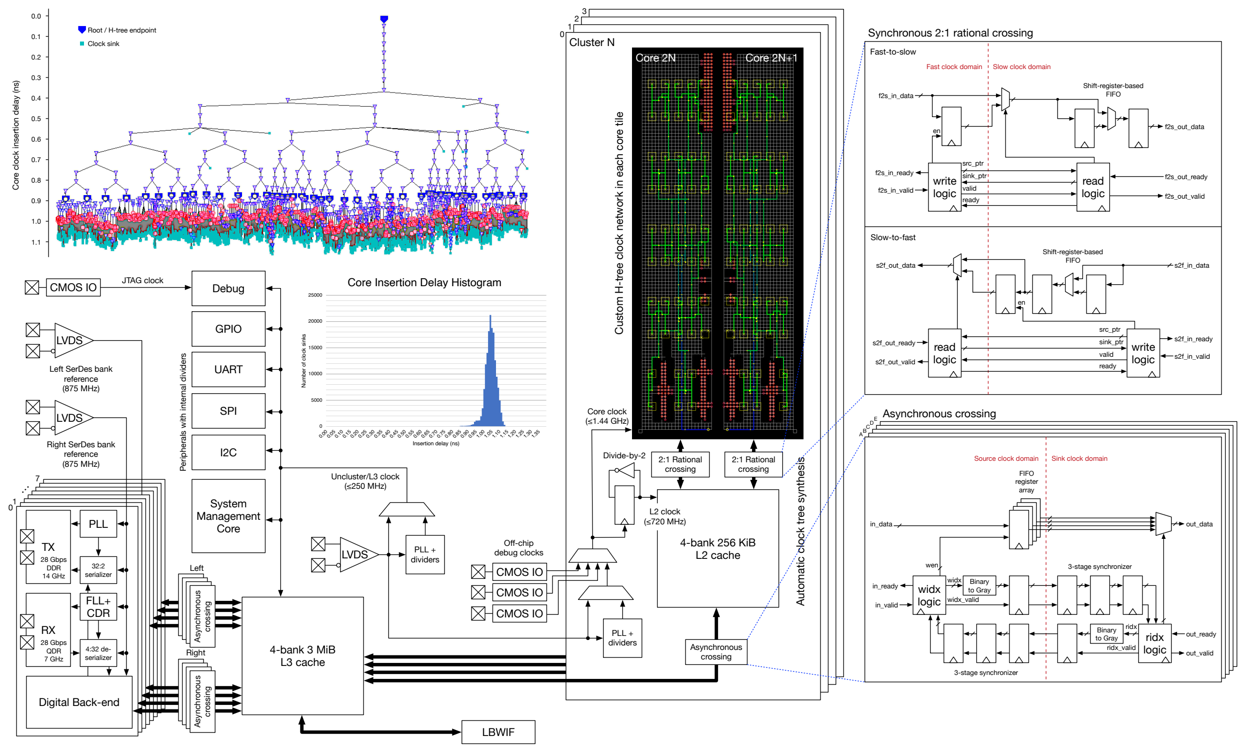


Figure 3.13: Eagle clocking diagram.

The tall and narrow aspect ratio of the tiles causes automatic clock tree synthesis (CTS) to introduce significant unwanted clock skew from the bottom to the top of the tile, because the clock input pin is placed along the bottom edge with the other pins. To combat this clock skew, a semi-custom H-tree is implemented using the Innovus place-and-route tool, as shown in Figure 3.13. This technique involves placing specific clock buffers in a fractal-like H pattern up to a certain number of levels; on Eagle, the tile H tree has six levels. After this, the placement tool inserts multiple smaller clock trees rooted at the H-tree sinks. The clock sink insertion delay histogram in Figure 3.13 shows the distribution of insertion delays across flip flop sinks in the design. The insertion delay has a mean of 1040 ps and a standard deviation of 37.8 ps, meaning that 99% of cells ($\pm 3\sigma$) are within 227 ps or 24% of the 1.05 GHz (952 ps) clock period. This is not particularly impressive, but it is significantly better than the roughly one nanosecond of skew present without H-trees. A square tile floorplan would resolve this issue, but the decoupled nature of the Hwacha unit, which is placed at the top side of the tile, allows the design to tolerate some clock skew.

The insertion delay becomes more significant at the tile-cluster boundary. Because the tile is a hierarchical module within the cluster, the tile insertion delays are added to the insertion delay to the tile clock pin within the cluster. This adds significant skew between logic clocked in the tile and cluster, which can be as large or larger than the desired clock period, destroying any chance of closing timing. Such problems are difficult to fix as they necessitate re-implementing the tiles with new I/O timing constraints to force the tile optimizer to reduce the insertion delay. By design, these problems are avoided by ensuring that logic driven by the tile is sunk directly into a flip flop with no combinational logic in between. In the Eagle design, this flip flop is part of a 2:1 ratiosynchronous crossing, which allows the L2 logic to be clocked at half the frequency of the tile. This issue does not exist at the cluster-uncluster interface as this interface is asynchronous, as is the interface between the serial link lanes and the uncluster.

The 14 GHz (TX) and 7 GHz (RX) high-frequency serial link clocks are synthesized by phased-locked loops (PLLs) and frequency-locked loops (FLLs), respectively, from an 875 MHz reference as shown in Figure 3.13. These loops contain LC tanks that are internal to the IP and digital logic that is placed-and-routed. The high-frequency clocks are divided and used to clock the digital serial link back-end. This creates a source-synchronous interface at each serial link macrocell which must be characterized by the designer. While BAG does not natively support this procedure, Eagle used a manual characterization flow with some additional guardband to ensure no timing violations occur at the serial link interfaces

The Eagle PLL design is based on a similar PLL architecture [90] used in the Splash2 and Hurricane chips. The DCO supply is regulated by a BAG-generated low-dropout regulator with an N-channel pass element to attenuate supply noise, which requires a 1.8 V supply to provide enough supply voltage headroom. This adds complexity to the power grid, which does not have 1.8 V available across the entire chip. To combat this, a local 1.8 V power island is connected to bumps on the interior of the chip to provide access to 1.8 V. The power strap arrangement for the region containing both the nominal 0.8 V supply and 1.8 V supply is shown in Figure 3.14.

3.1.4.3 Interconnect design

Each cluster has a set of TileLink primary ports and a single TileLink secondary port. The secondary port provides access to the configuration and status registers within the L2 cache and conforms to the uncached lightweight (TL-UL) level. The primary ports all use the cached variety of TileLink (TL-C), which is required for the clusters to be coherent. Each cluster has 5 TL-C ports, one for each L3 memory bank and an additional for connecting to the system bus (sbus). This fully connected topology is known as a crossbar, and the Eagle configuration is illustrated in Figure 3.15. Each TL-C connection is configured to have 128 bits of data and each TL-UL connection is configured to have 64 bits of data. Four of the TL-C channels can carry data (A, B, C, D), while only two TL-UL can do so (A, D). The B channel of the cluster-to-sbus connection is optimized away by the synthesis tool, as the sbus has no cache behind it and therefore cannot initiate any coherency transactions itself.

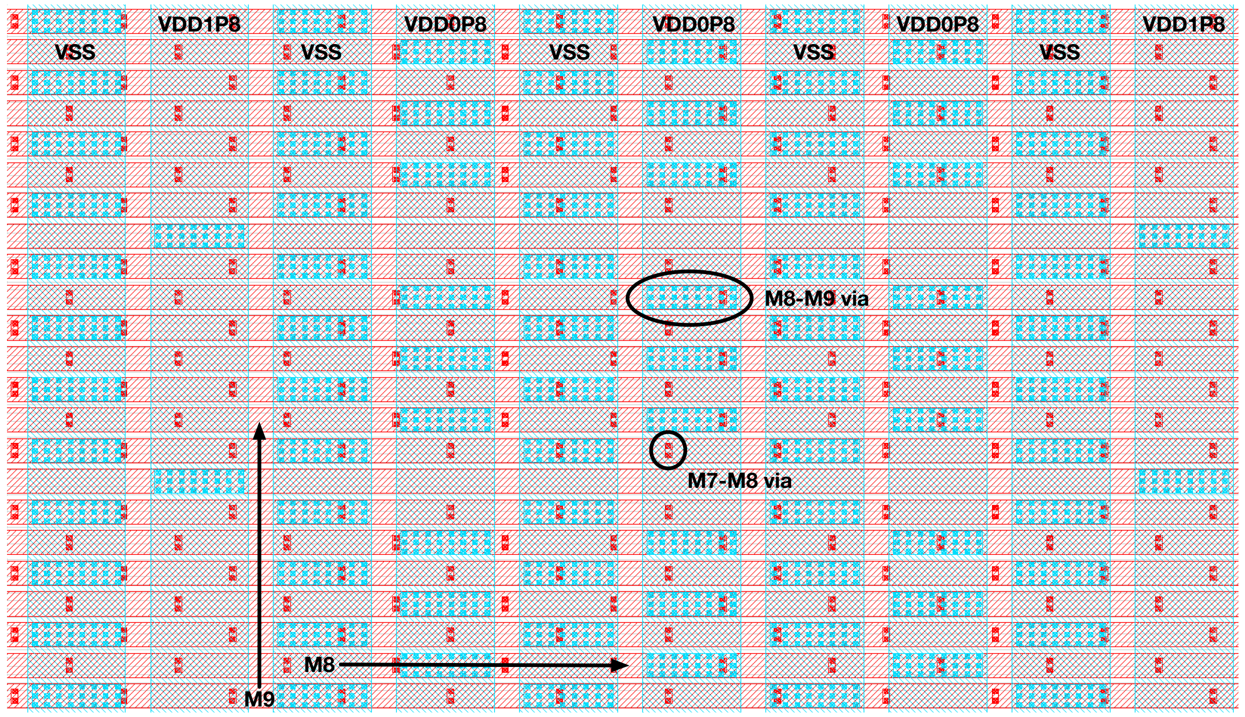


Figure 3.14: An example of multi-voltage power straps on EagleX using a 1:3 supply ratio. Only metal 8 (M8, horizontal) and metal 9 (M9, vertical) are shown.

Therefore, there are a total of $16 \cdot 4 \cdot 128 + 4 \cdot 3 \cdot 128 + 4 \cdot 2 \cdot 64 = 10240$ data wires crossing between the clusters and the L3 cache and system bus. This does not include address wires or other metadata, which account for a significant fraction of the total size of the bus. This number depends highly on the configuration of the bus and the overall SoC, but it is roughly 40% of the data bus size. This results in over 14,000 wires. Assuming a minimum vertical metal pitch of 50 nm, this corresponds to 0.7 mm of horizontal space required to route this many wires vertically within the crossbar.

This number is large; it is on the order of the die length, and larger than the cluster floorplan width. This logic limited the maximum clock frequency of the L3 cache to 250 MHz in place-and-route, but runs with more aggressive bus configurations did not even finish routing successfully. Figure 3.16 shows that the area between the clusters and the L3 cache does not have high placement density, while Figure 3.17 shows significant routing congestion in this area[7].

For designs of this size, networks-on-chip [91, 92] may be more appropriate topologies than crossbars, as they reduce the number of wires by requiring some paths to take multiple "Hops" to route. This increases the latency of some transactions, but generally allows the crossbar clock frequency to increase, resulting in a net throughput improvement. Another approach is to use DNN-based routing congestion prediction [93] to predict the congestion

[7]This congestion map is from a run that was not taped out.

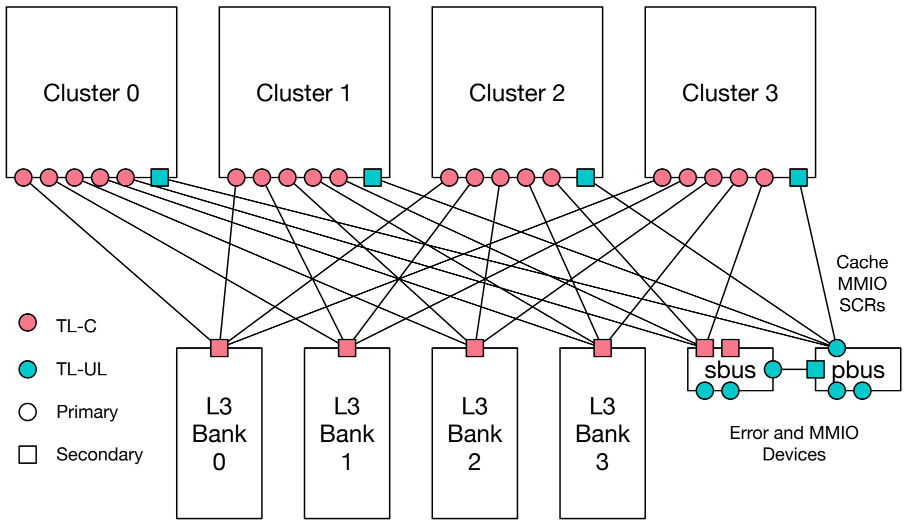


Figure 3.15: Eagle crossbar topology.

issues before running placement. This does not solve the hardware architecture problem, but gives early feedback to the designer to allow the designer to make changes earlier in the development cycle.

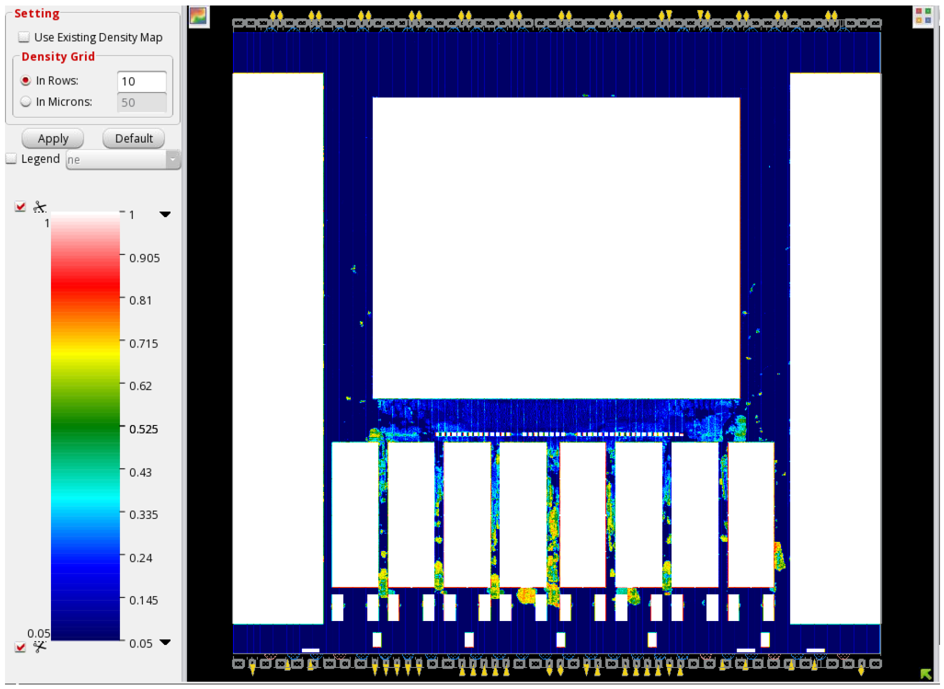

Setting
[ ] Use Existing Density Map
Density Grid
[✓ In Rows:] 10
In Microns: 50
Apply Default
[ ] Legend ne

Figure 3.16: Low placement density in area with high routing density on Eagle.

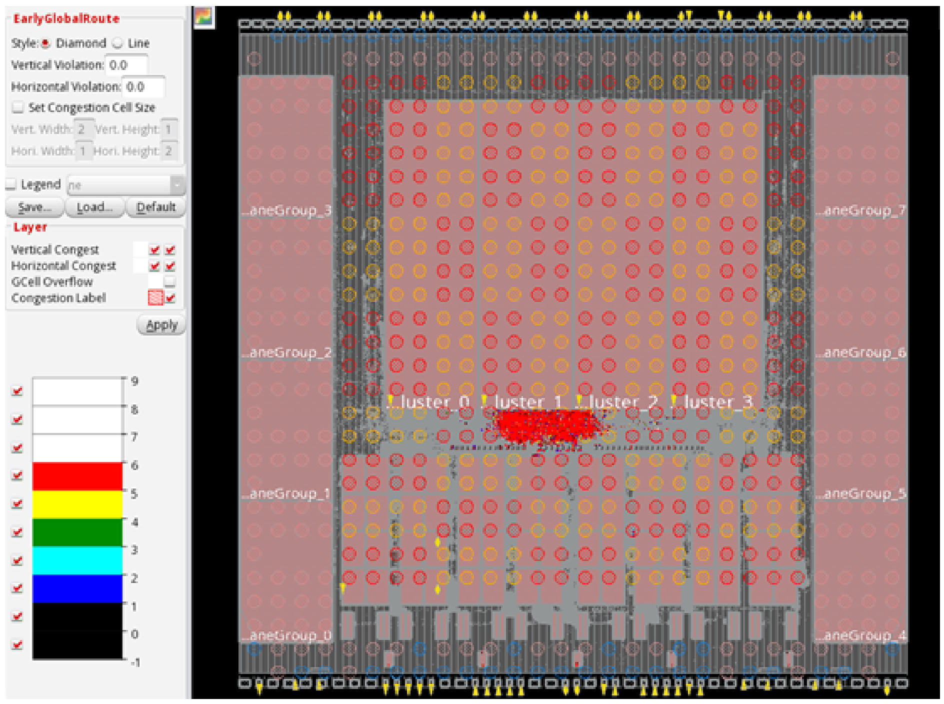

EarlyGlobalRoute
Style: [✓ Diamond] Line
Vertical Violation: 0.0
Horizontal Violation: 0.0
[ ] Set Congestion Cell Size
Vert. Width: 2 Vert. Height: 1
Hori. Width: 1 Hori. Height: 2
[ ] Legend ne
Save... Load... Default
Layer
Vertical Congest ✓ ✓
Horizontal Congest ✓ ✓
GCell Overflow
Congestion Label ✓
Apply

Figure 3.17: Example of routing congestion in a large crossbar on Eagle.

3.2 EagleX: A 21-core Generated RISC-V SoC

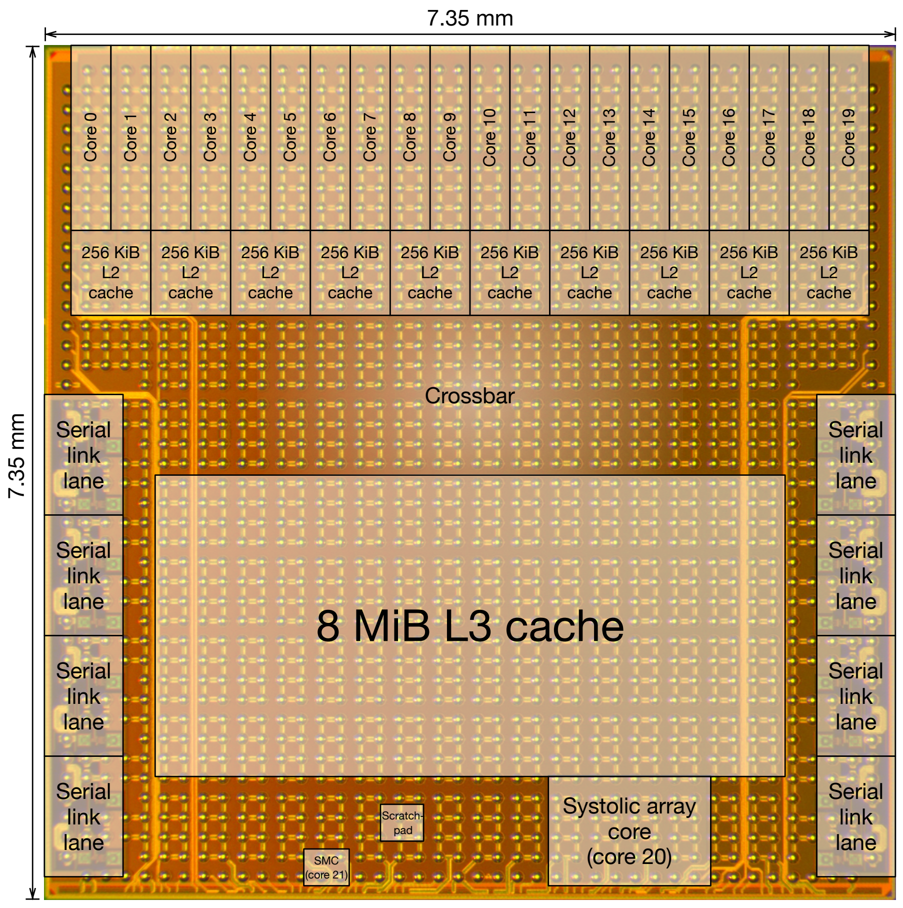


Figure 3.18: Annotated EagleX die micrograph.

3.2.1 Background

Eagle effectively demonstrates the efficiency and flexibility of vector accelerators for edge machine learning applications, but its incapability of running an operating system limits its utility. EagleX[8]expands upon the Eagle architecture with updated vector accelerators and a memory system that can boot a multicore operating system like Linux. Additionally, the Hammer design methodology co-developed with Eagle allows for easy design space exploration of new accelerator IP like systolic arrays. Systolic arrays, first described in 1982 [94], have seen a surge in popularity because of the ease with which dense two-dimensional compute patterns like deep neural networks (DNNs) map to their spatial structure.

[8]Eagle was co-led by the author and Colin Schmidt. Zhongkai Wang led the serial link development with the help of others. SiFive donated the multi-level cache generator.

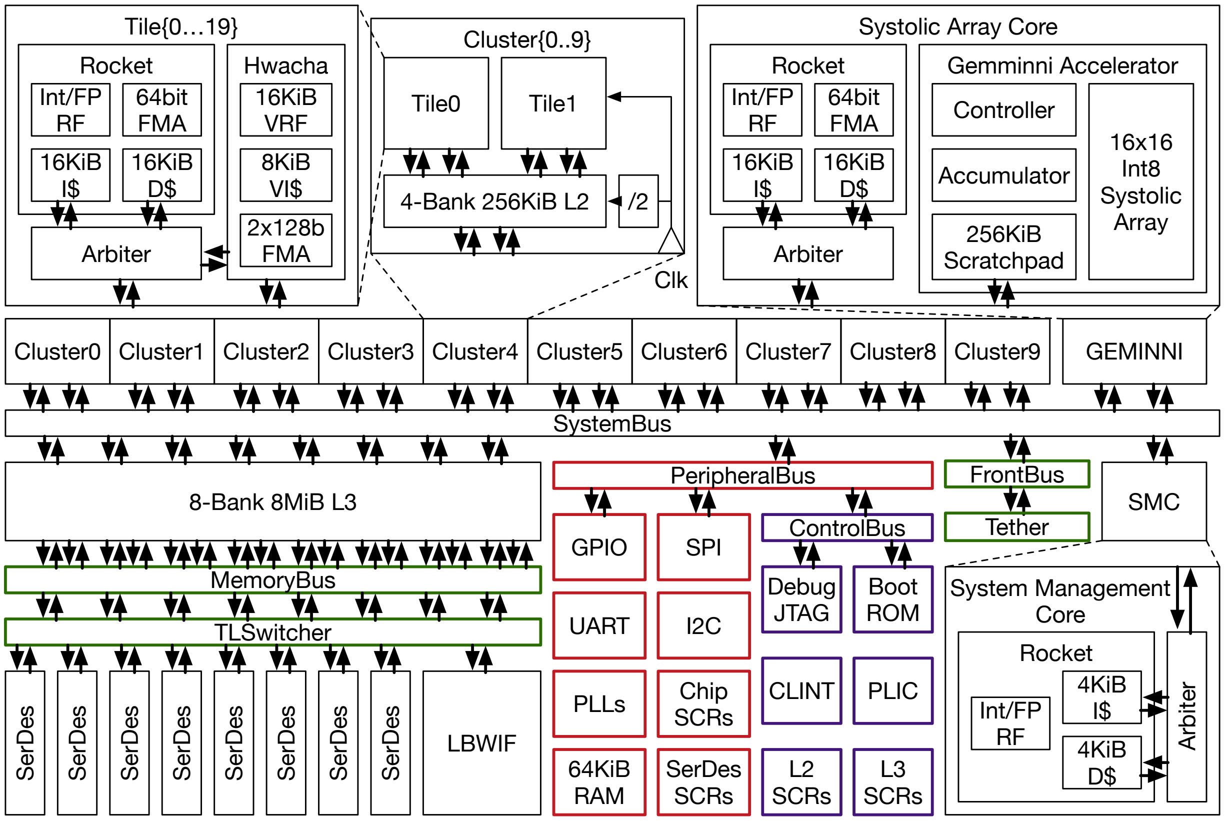


Figure 3.19: EagleX block diagram.

3.2.2 Architecture

The EagleX system-on-chip is manufactured in TSMC 16nm FinFET and measures 7.35 mm by 7.35 mm[9], as shown in Figure 3.18. The EagleX architecture is presented in Figure 3.19 and closely follows the Eagle architecture, but with some noteworthy deviations. EagleX has twenty in-order RV64GC applications cores built with the Rocket core generator and configured to include separate 16 KiB instruction and data caches, a memory management unit (MMU), and privilege levels. Each core has a dedicated Hwacha vector accelerator which uses a non-standard Hwacha extension and includes additional integer fused-multiply add (FMA), truncation, merge, max, and min instructions not implemented on Eagle. The twenty applications cores are organized into ten clusters, with two cores sharing a 256 KiB, 2-bank, inclusive, directory-based L2 cache. Each L2 cache runs at half the frequency of its cores and communicates across a ratiosynchronous 2:1 crossing. Each cluster on the chip can be clocked independently from one of three separate PLL clock sources or their references. Unlike Eagle, the clusters are not in independent power domains, which reduces the complexity of the on-die power grid.

[9]In TSMC's 16FFC process, 7.35 mm is 7.5 mm in drawn dimensions with a 98% shrink factor.

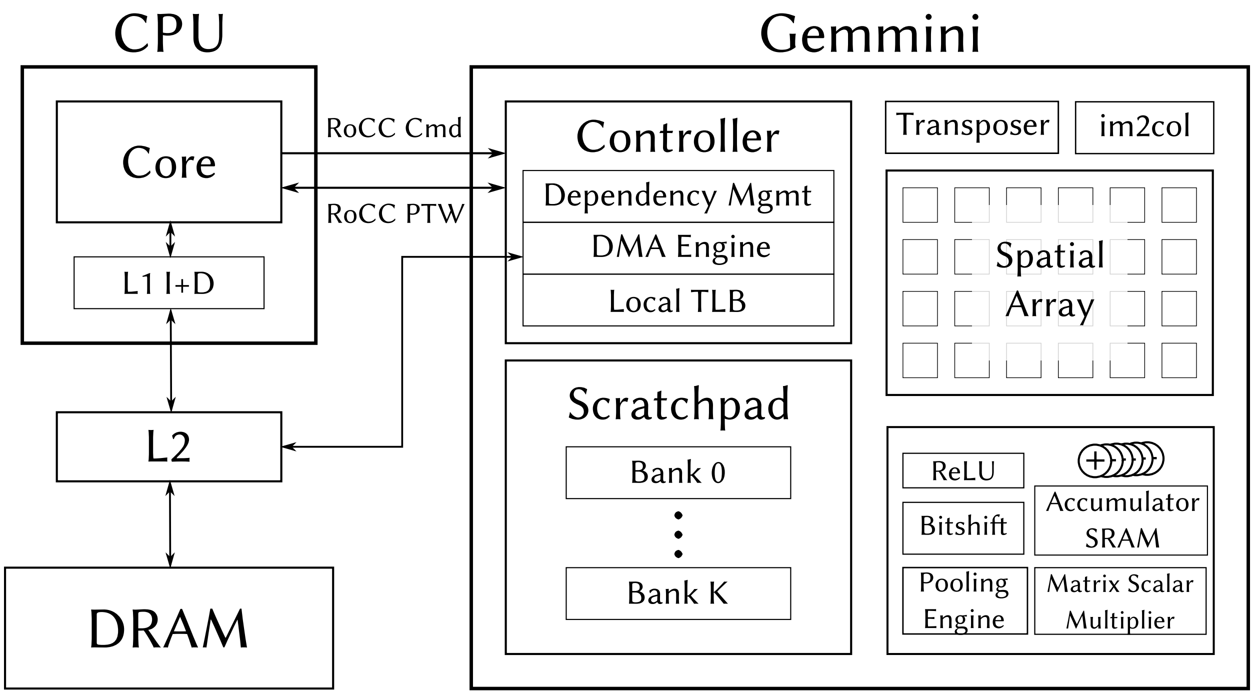


Figure 3.20: EagleX systolic array accelerator architecture [9].

EagleX includes two additional differently configured Rocket cores in the system. The first is a system management core, like Eagle, used to manage system boot, handle clock selection, and perform other system management tasks This core implements the RV64IMAC, which is a reduced feature-set ISA variant from RV64GC without floating point support, and has separate 4 KiB instruction and data caches. The core has no L2 cache and is instead backed directly by the L3 cache via the system bus.

The final Rocket core in the EagleX system has the same base configuration as the twenty main applications cores. However, this core includes a new systolic array accelerator, Gemmini [9], targeted at processing deep neural networks (DNNs) energy-efficiently. Like Hwacha, Gemmini uses the Rocket custom co-processor (RoCC) interface to extend the base RISC-V ISA with custom instructions and a decoupled microarchitecture. The architecture of this core and accelerator is shown in Figure 3.20. This core has no L2 cache and is backed directly by the L3 cache. Instead, the Rocket configuration includes a 256 KiB scratchpad for use by Gemmini, which can improve the utilization of the systolic array versus an L2 cache because it never has to handle misses. The systolic array includes 64 KiB of accumulator memory for storing partial sums. The inclusion of heterogeneous compute accelerators allows easy direct comparison of the two accelerator architectures.

The EagleX system-on-chip includes an 8 MiB, 16-bank, inclusive, directory-based L3 cache that is shared among all twenty clusters, the system controller, and the systolic array core. The L3 is backed by either a low speed off-chip interface or eight high speed serial links, as on the Eagle SoC. EagleX also includes GPIOs, a UART, SPI, I^{2}C, an on-chip

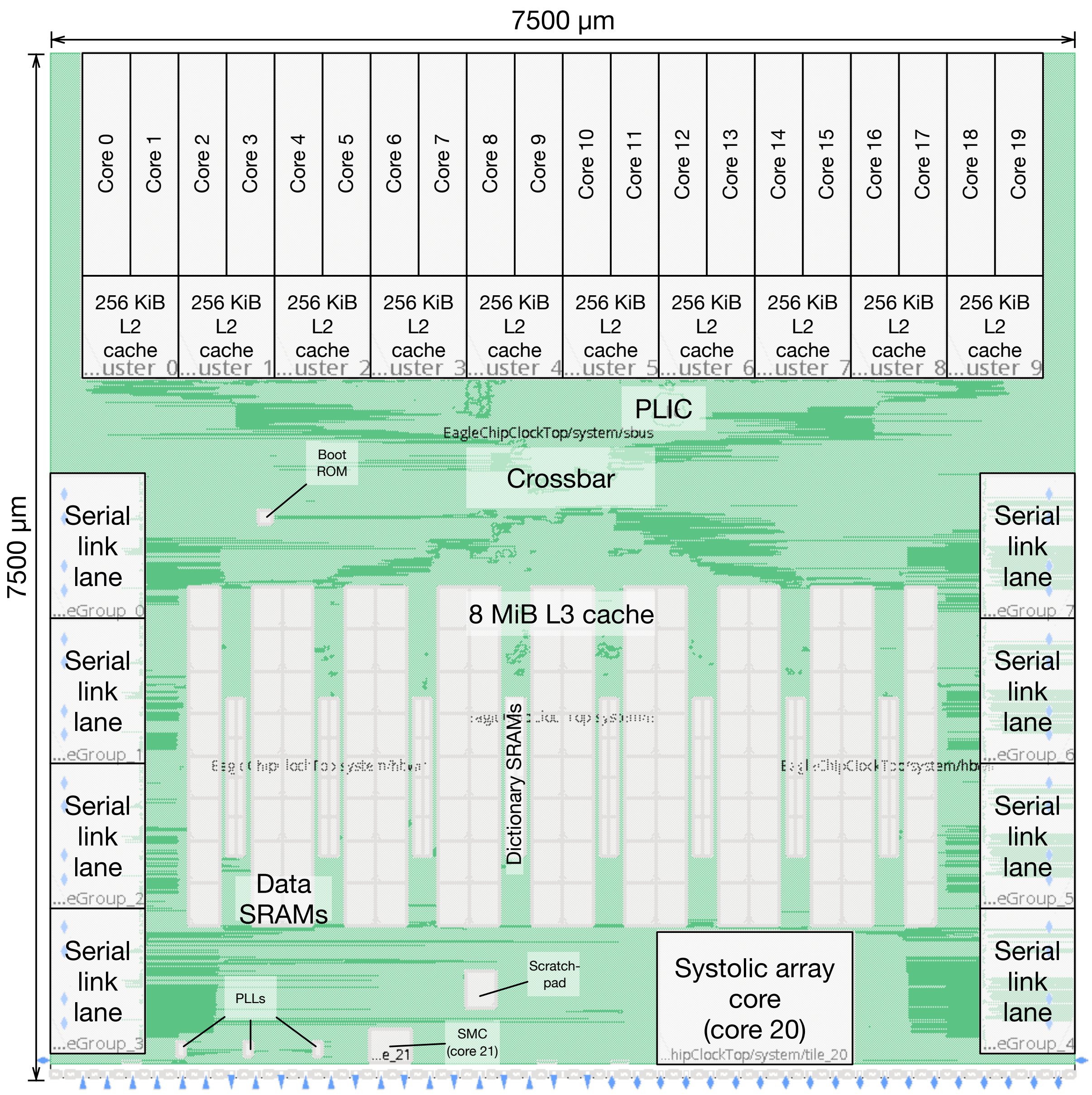


Figure 3.21: EagleX chip floorplan (hierarchical).

memory-mapped SRAM bank, and a boot ROM.

3.2.3 Physical design challenges

At 54 mm^2, EagleX is more than double the area of Eagle and just over half the size of the Apple A9 [95], the first iPhone SoC to use TSMC 16nm FinFET. Because of this, appropriate management of hierarchy and area are even more critical on EagleX than on

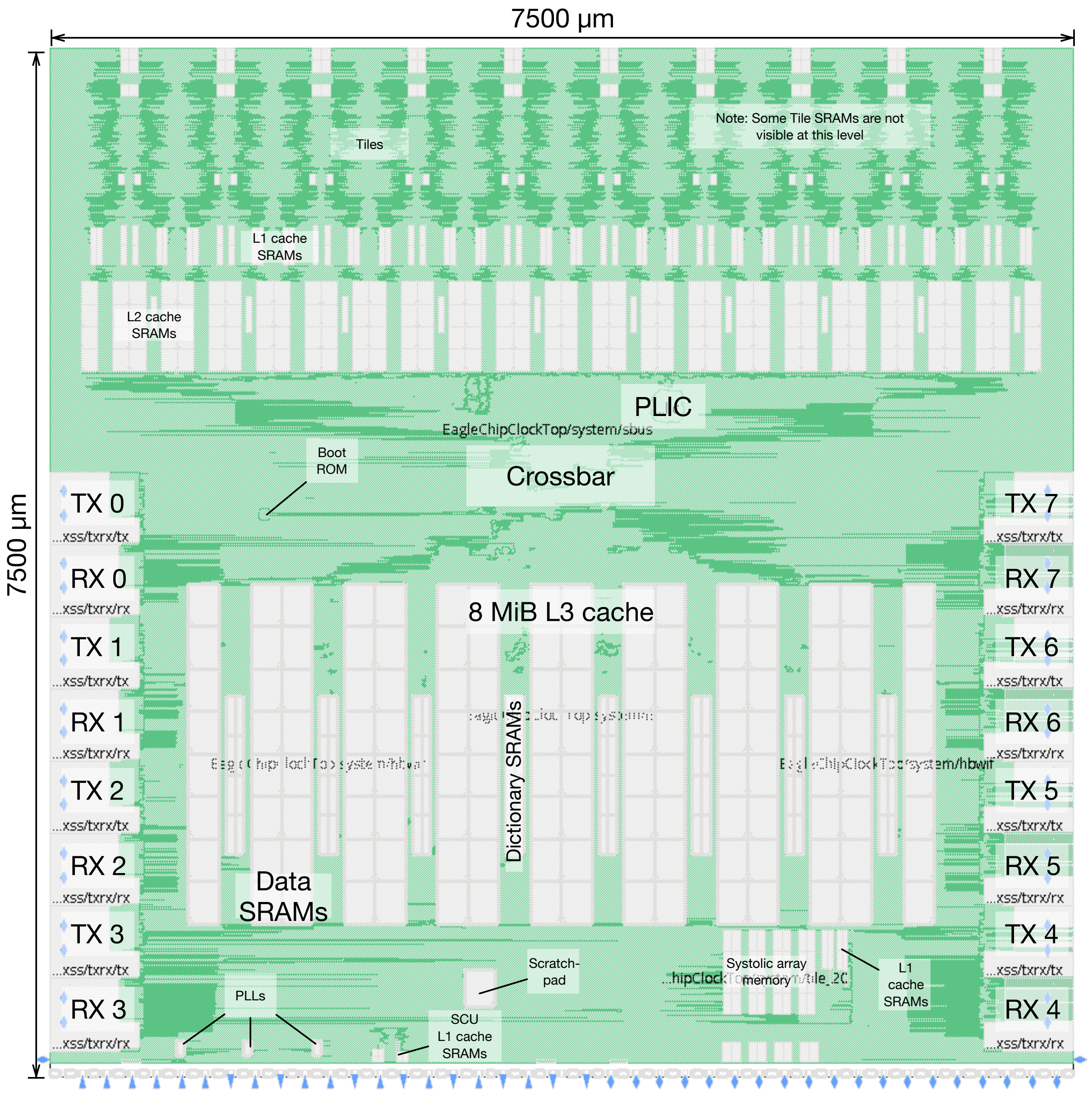


Figure 3.22: EagleX chip floorplan (flat).

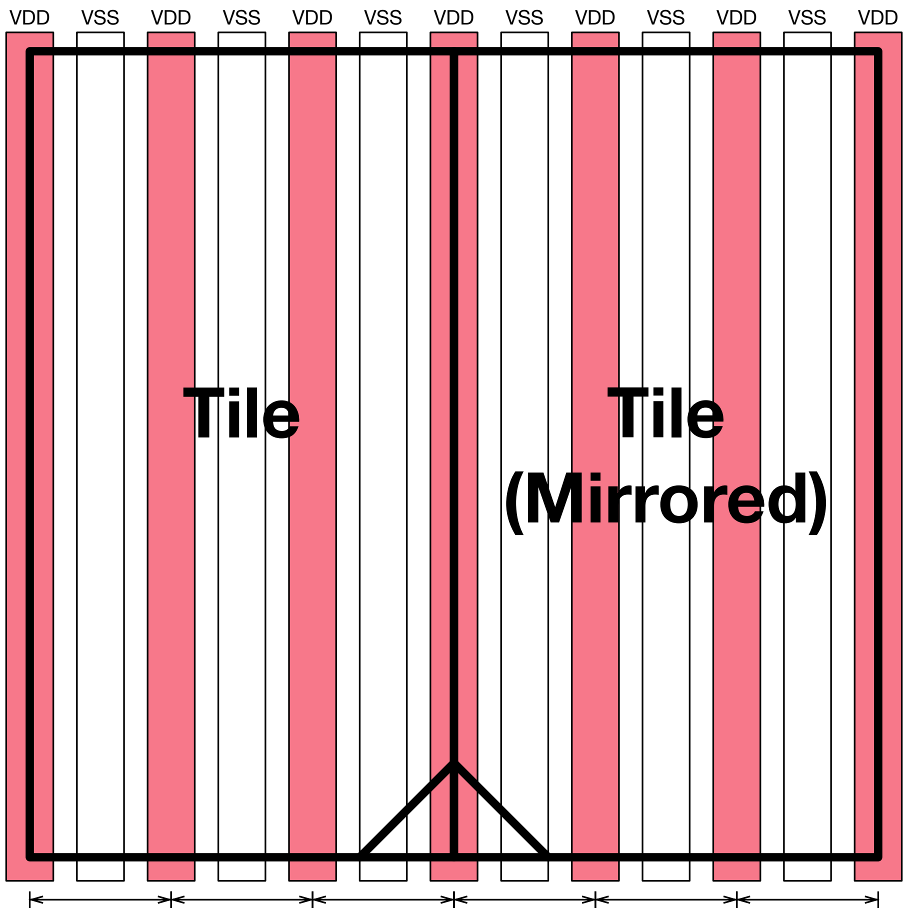


Figure 3.23: Power strap alignment on mirrored tiles.

Eagle. Fortunately, the physical design automation developed for the Eagle project enables subsequent projects like EagleX to reuse significant portions of the implementation flow and avoid many early pitfalls in the design process. The main applications tiles and cluster used very similar floorplans to their Eagle counterparts, so it is unsurprising that these have significantly fewer up-front physical design challenges than during Eagle development. However, the systolic array is completely new IP, taped out for the first time on EagleX. Using technology-specific hooks in Hammer, discussed in Chapter 4, the systolic array was able to be implemented with dramatically fewer timing and design rule violations early in the physical design process. This floorplan is entirely new to EagleX, including the Rocket core floorplan, which, despite being similarly configured to the applications cores, has a different aspect ratio due to the topology of the systolic array accelerator. This motivates the need for floorplan generators that can produce ideal floorplans in multiple aspect ratios.

While the crossbar is significantly larger and has more endpoints than the one on Eagle, the additional area, improved floorplan, and RTL optimizations led to significantly easier

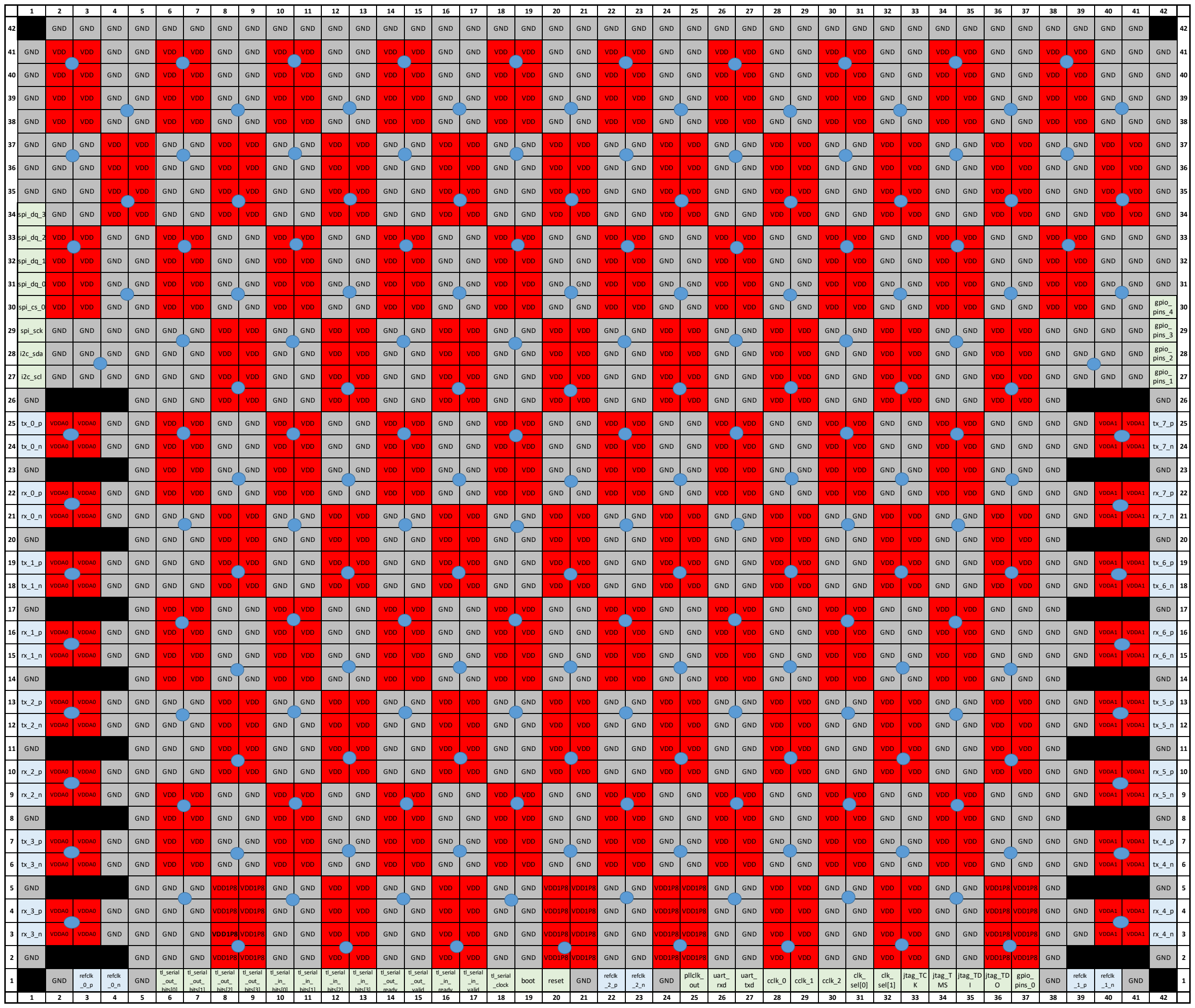
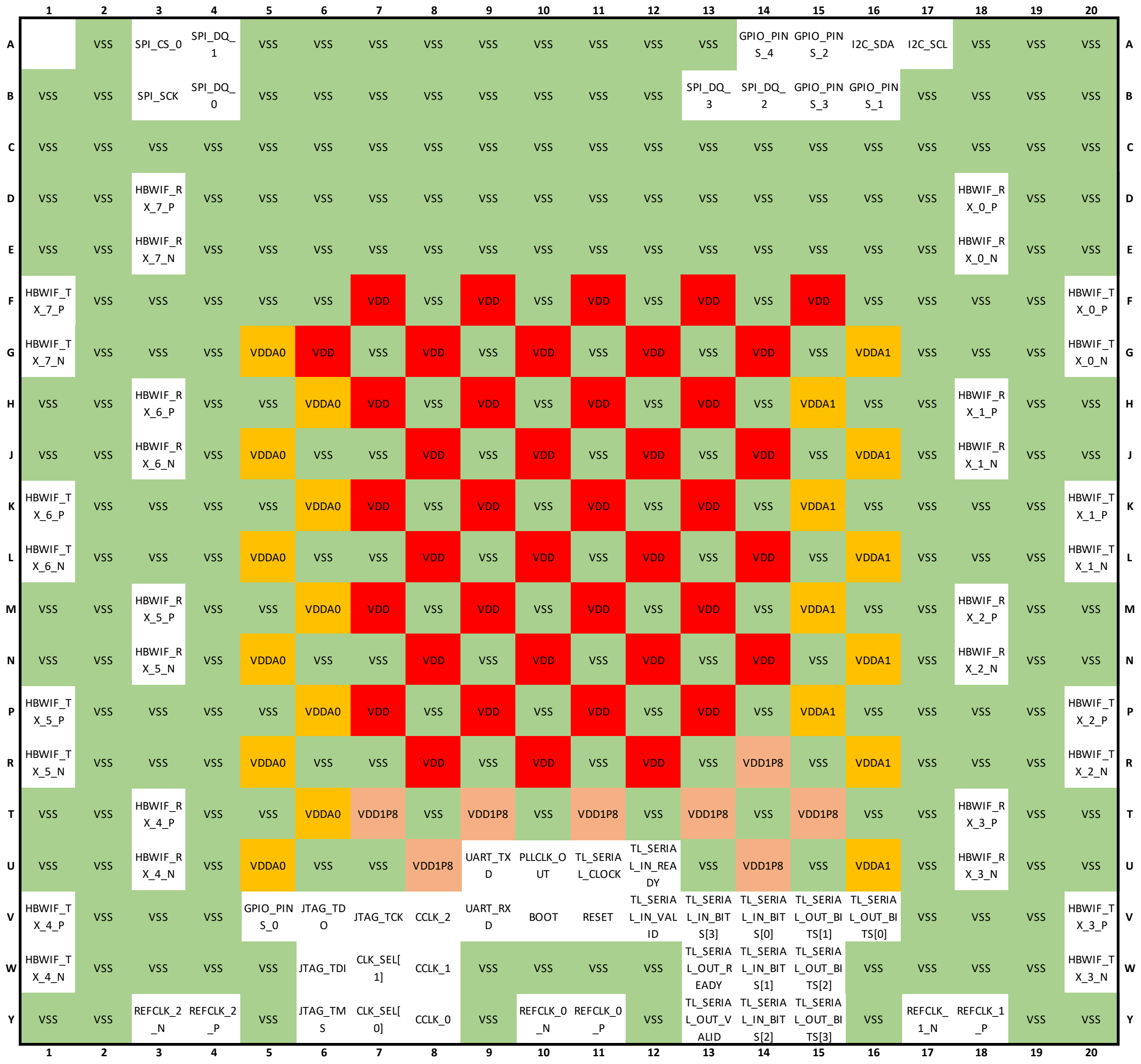
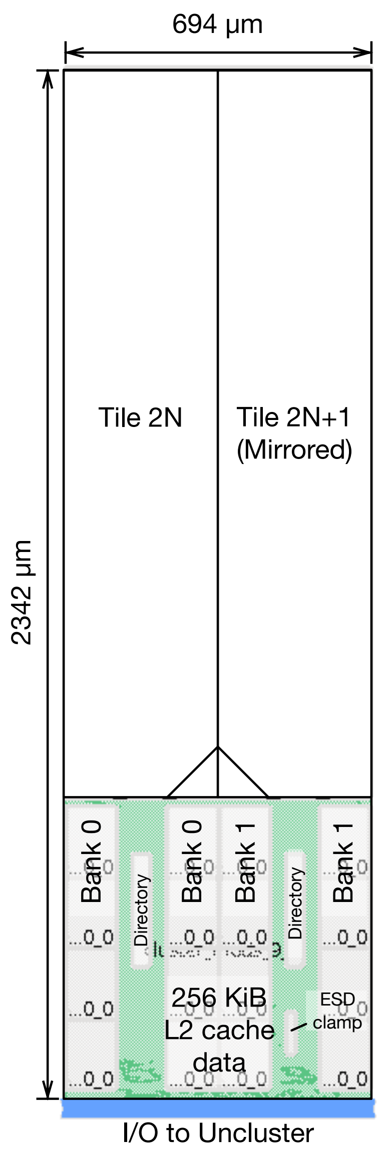


Figure 3.24: EagleX bump designations. A blue circle indicates a potential location for a via-in-pad if no package is used.

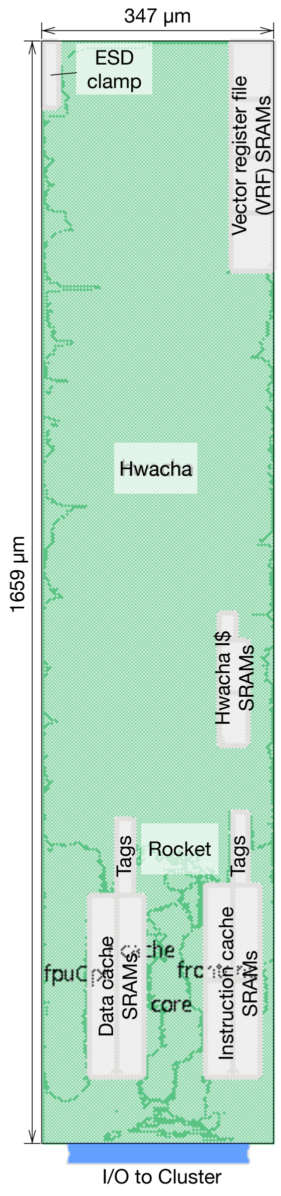


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
A		VSS	SPI_CS_0	SPI_DQ_1	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	GPIO_PINS_4	GPIO_PINS_2	I2C_SDA	I2C_SCL	VSS	VSS	VSS	A
B	VSS	VSS	SPI_SCK	SPI_DQ_0	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	SPI_DQ_3	SPI_DQ_2	GPIO_PINS_3	GPIO_PINS_1	VSS	VSS	VSS	VSS	B
C	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	C
D	VSS	VSS	HBWIF_RX_7_P	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	HBWIF_RX_0_P	VSS	VSS	D
E	VSS	VSS	HBWIF_RX_7_N	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	VSS	HBWIF_RX_0_N	VSS	VSS	E
F	HBWIF_TX_7_P	VSS	VSS	VSS	VSS	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VSS	VSS	VSS	HBWIF_TX_0_P	F
G	HBWIF_TX_7_N	VSS	VSS	VSS	VDDA0	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	VSS	VSS	HBWIF_TX_0_N	G
H	VSS	VSS	HBWIF_RX_6_P	VSS	VSS	VDDA0	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	VSS	HBWIF_RX_1_P	VSS	VSS	H
J	VSS	VSS	HBWIF_RX_6_N	VSS	VDDA0	VSS	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	HBWIF_RX_1_N	VSS	VSS	J
K	HBWIF_TX_6_P	VSS	VSS	VSS	VSS	VDDA0	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	VSS	VSS	VSS	HBWIF_TX_1_P	K
L	HBWIF_TX_6_N	VSS	VSS	VSS	VDDA0	VSS	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	VSS	VSS	HBWIF_TX_1_N	L
M	VSS	VSS	HBWIF_RX_5_P	VSS	VSS	VDDA0	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	VSS	HBWIF_RX_2_P	VSS	VSS	M
N	VSS	VSS	HBWIF_RX_5_N	VSS	VDDA0	VSS	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	HBWIF_RX_2_N	VSS	VSS	N
P	HBWIF_TX_5_P	VSS	VSS	VSS	VSS	VDDA0	VDD	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDDA1	VSS	VSS	VSS	VSS	HBWIF_TX_2_P	P
R	HBWIF_TX_5_N	VSS	VSS	VSS	VDDA0	VSS	VSS	VDD	VSS	VDD	VSS	VDD	VSS	VDD1P8	VSS	VDDA1	VSS	VSS	VSS	HBWIF_TX_2_N	R
T	VSS	VSS	HBWIF_RX_4_P	VSS	VSS	VDDA0	VDD1P8	VSS	VDD1P8	VSS	VDD1P8	VSS	VDD1P8	VSS	VDD1P8	VSS	VSS	HBWIF_RX_3_P	VSS	VSS	T
U	VSS	VSS	HBWIF_RX_4_N	VSS	VDDA0	VSS	VSS	VDD1P8	UART_TXD	PLLCLK_OUT	TL_SERIAL_CLOCK	TL_SERIAL_IN_READY	VSS	VDD1P8	VSS	VDDA1	VSS	HBWIF_RX_3_N	VSS	VSS	U
V	HBWIF_TX_4_P	VSS	VSS	VSS	GPIO_PINS_0	JTAG_TDO	JTAG_TCK	CCLK_2	UART_RXD	BOOT	RESET	TL_SERIAL_IN_VALID	TL_SERIAL_IN_BITS[3]	TL_SERIAL_IN_BITS[0]	TL_SERIAL_OUT_BITS[1]	TL_SERIAL_OUT_BITS[0]	VSS	VSS	VSS	HBWIF_TX_3_P	V
W	HBWIF_TX_4_N	VSS	VSS	VSS	VSS	JTAG_TDI	CLK_SEL[1]	CCLK_1	VSS	VSS	VSS	VSS	TL_SERIAL_OUT_READY	TL_SERIAL_IN_BITS[1]	TL_SERIAL_OUT_BITS[2]	VSS	VSS	VSS	VSS	HBWIF_TX_3_N	W
Y	VSS	VSS	REFCLK_2_N	REFCLK_2_P	VSS	JTAG_TMS	CLK_SEL[0]	CCLK_0	VSS	REFCLK_0_N	REFCLK_0_P	VSS	TL_SERIAL_OUT_VALID	TL_SERIAL_IN_BITS[2]	TL_SERIAL_OUT_BITS[3]	VSS	REFCLK_1_N	REFCLK_1_P	VSS	VSS	Y
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	

Figure 3.25: EagleX BGA package pinout.



(a) Cluster



(b) Tile

Figure 3.26: EagleX cluster and tile floorplans. Dimensions shown are as drawn before a 98% optical shrink.

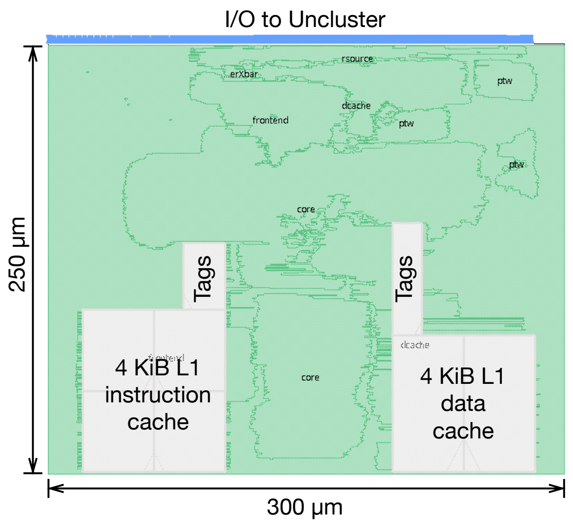


Figure 3.27: EagleX system management core floorplan.

timing closure and routing than on Eagle. The additional amount of SRAM and number of instances and wires in the uncluster of EagleX increased tool runtime significantly, but improved hierarchical flow automation in Hammer reduced the total number of top-level runs that were necessary throughout the construction of EagleX.

3.2.3.1 Floorplan

The top-level hierarchical EagleX floorplan is shown in Figure 3.21. Figure 3.22 shows a flattened floorplan, where all levels of the hierarchy are visible at once. In this view, the mirrored instances of the tiles are clearly visible. Mirroring the tiles allows the SRAMs in the vector unit and L1 caches to be abutted at the boundary, which maximizes the available horizontal span within the narrow tile floorplan. Consequently, this mirroring requires that the tile width be a multiple of the vertical power strap group pitch and that the vertical power strap x-offset be one half of the power strap width so that straps of the same net will overlap. This requirement is demonstrated in Figure 3.23. The L3 cache bank count is increased to 16 from 4 on Eagle to improve memory bandwidth off-chip. However, the L2 cache bank count is reduced to 2 from 4 on Eagle to reduce the amount of wiring between the L2 and L3 caches.

The ten clusters and twenty tiles are placed along the top edge of the chip. Unlike Eagle, there is sufficient area to place all chip I/O cells along the bottom edge, which allows the tiles and clusters to be placed along the top edge, allowing for more optimal die utilization and fewer long routes. To accommodate the preferred geometry of the systolic array accelerator, the systolic array core is arranged in a more square aspect ratio than the applications cores. To avoid blocking the routing between the cluster L2 caches and the L3 cache, the systolic array core is placed along the bottom edge of the chip. The system control core and the scratchpad SRAM are also placed along the bottom edge. The PLLs are placed physically close to their differential clock receivers to reduce jitter from noise in the clock network. They are grouped closely together to avoid the requirement to have a 1.8V supply across large portions of the die. Although EagleX is packaged, the supply bumps are grouped in gangs of eight, like Eagle, to allow chip-on-board if necessary. The bump designations on the die are shown in Figure 3.24, and the BGA package pins are shown in 3.25.

The tile, cluster, and system management core (SMC) floorplans are shown in Figure 3.26.b, Figure 3.26.a, and Figure 3.27, respectively. The serial link lane floorplan is similar to that on Eagle and can be seen in Figure 3.11.a. The systolic array core floorplan is shown in Figure 3.28. In the floorplan, the systolic array accelerator is placed in the left half of the cell, with the SRAMs spread out to allow room for the routing to the systolic processing elements (PEs), which are placed and routed automatically below and to the left of the systolic array SRAMs. This placement produces satisfactory results, but a custom-placed regular structure would improve QoR and reduce overall area [96]. The Rocket CPU core is placed towards the rightmost edge of the cell along with the pins, with the L1 cache SRAMs placed in the top right corner. This arrangement essentially creates a topology that is similar to the Hwacha tiles rotated by ninety degrees, as both the RoCC accelerators should be placed far away from the pins because they communicate through the Rocket core. This stresses the importance of understanding the logical design when floorplanning.

3.2.4 Results and future work

At the time of this writing, testing of EagleX is ongoing. The use of a package improves assembly yield significantly, with all assembled boards functioning normally. Preliminary testing shows that the physical design bug in the L2-L3 cache interface found on Eagle is not present on EagleX, which enables multicore operating systems to boot and run parallel workloads. The chip has been verified to boot Linux on the twenty applications cores, as shown in Listing 3.1. The system control unit and the systolic array accelerator core have yet to be tested

The primary objective in testing EagleX is to demonstrate the energy efficiency of the vector units when running a real edge neural net workload, like SqueezeNet [97] or SqueezeNext [98]. Work is ongoing to use ONNX[10] to map neural net workloads to Hwacha and Gemmini for independent and comparative testing.

[10]https://onnx.ai/

```
# cat /proc/cpuinfo
processor       : 0
hart            : 0
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0

processor       : 1
hart            : 1
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0

processor       : 2
hart            : 16
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0

...

processor       : 19
hart            : 15
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0
```

Listing 3.1: EagleX running Linux on the twenty main applications cores.

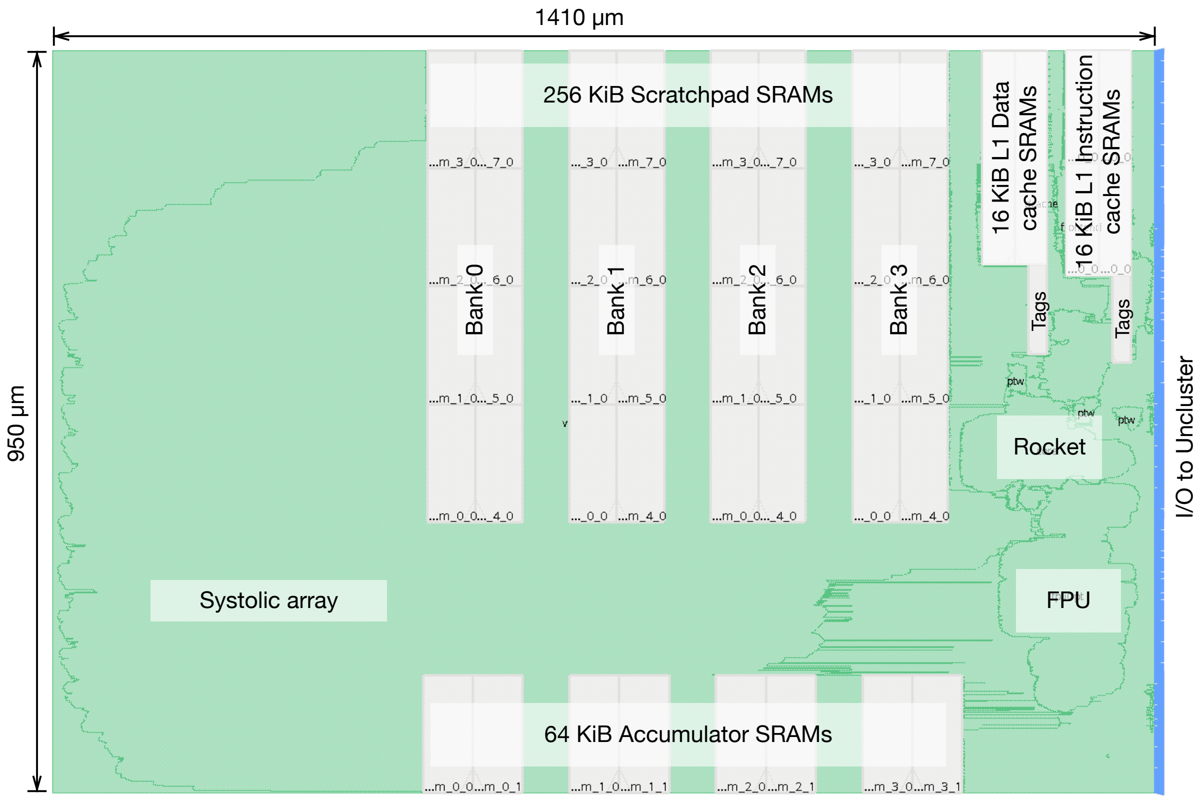


Figure 3.28: Floorplan of the EagleX systolic array accelerator with its CPU core.

Chapter 4

Hammer: A Physical Design Generator Platform

4.1 Motivation

Traditional physical design flows typically require users to combine multiple orthogonal sets of information into a single stream of tool commands and specifications in a manner that decreases reusability of the flow, resulting in longer design cycles, higher non-recurring engineering (NRE) expenses, and lower designer productivity. This effect is more noticeable in generator-based design flows, which, by Amdahl's Law, are limited in their maximum productivity improvement by such long-latency manual tasks. Despite the recognition of this challenge by academic and industrial ASIC design teams [99], progress is stymied by a lack of shared development between teams isolated at their respective institutions. To combat this, an open-source EDA tool chain, OpenROAD [100], has been developed, aiming to lower barriers to entry for hardware designers. While commercial EDA tool flows currently outperform their open-source counterparts, the democratization of ASIC design flows afforded by open-source movements is critical for the hardware design community to achieve the productivity levels that software designers have been enjoying for decades.

4.2 Hammer design philosophy

Hammer [1,2] [17] is an open-source physical design platform that aims to improve reusability of physical design components. Rather than implement the underlying EDA tools themselves, which is sufficiently covered by the OpenROAD project, Hammer implements a flow which can be used with any open-source or proprietary tool set. Hammer focuses on identifying and isolating all aspects of physical design inputs, categorizing each into one of three categories:

[1]The Hammer framework was initially developed by Edward Wang. The author contributed features for physical design, physical verification flows, hierarchical build flows, PCB design features, and other features.

[2]https://github.com/ucb-bar/hammer

1. Design-specific inputs, like I/O definitions, module hierarchy, and floorplan topology
2. Tool-specific inputs, like file formats, command syntax, and internal data flow
3. Technology-specific inputs, like routing layer information, standard cell libraries, and design rules

The Hammer development community refers to this as the "Separation of concerns." The guiding philosophy is to isolate all aspects of design intent into one of the three categories, when possible. Some types of design inputs, like the floorplan, invariably must contain two categories of input, as the floorplan must be driven by the logic in the specific design, but it also must take into account the sizing of SRAMs and the total gate area for the desired technology node.

Because of the complexity of ASIC design flows, it is frequently necessary to work around occasional bugs or optimize a troublesome design component. These are often done through esoteric EDA tool commands or scripts, which are usually tool- and technology-specific by nature. To enable this, "Incremental adoption" is a cornerstone design philosophy of Hammer. This essentially requires that any technique available to a traditional flow must be easily implementable within Hammer. However, this does not mean that Hammer supports a union of all possible EDA tool subroutines or commands. Instead, Hammer natively supports "Hooks," which allow the user to inject custom code at any point in the tool flow in a reusable manner. Once mature, and if appropriate, hooks can be brought into the core Hammer repository to improve reuse across designs.

4.3 Hammer design flow

The Hammer design flow is shown in Figure 4.1. The primary inputs to Hammer are Hammer IR, tool plugins written in Python, technology plugin hooks written in Python, and technology JSON files. The Hammer driver configures its internal database from a sequence of one or more Hammer IR files and then loads relevant tool and technology plugins dynamically in Python. These plugins augment or implement methods within the **HammerDriver** and **HammerTool** classes to build a custom tool flow and emit the tool commands that implement the design. In addition to the scripts required to drive the tools, Hammer emits a robust, configurable Makefile infrastructure tailored to the specific design hierarchy described in the Hammer IR. This technique allows extremely large designs to be implemented quickly without the need to set up complicated build flows. As hierarchical design has become ubiquitous for modern systems-on-chip, this feature makes Hammer very attractive for small academic and startup teams to ramp up quickly on complex designs.

4.3.1 Hammer IR

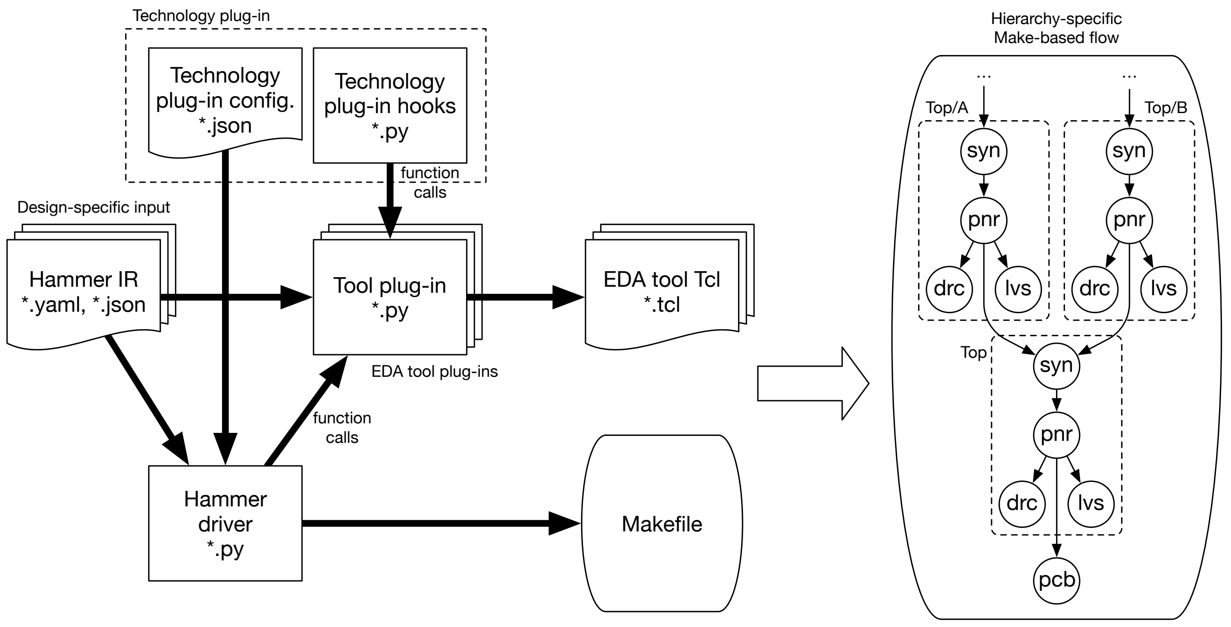


Figure 4.1: Hammer design flow.

Hammer uses a database format called Hammer IR for both input and output. Hammer IR can use either JSON[3] or YAML[4] for text-based serialization to files. JSON is commonly used for machine-generated output due to its rigidity, while YAML is preferred for hand-written files, as it is easier to read. An example of YAML-based Hammer IR is shown in Listing 4.1. Hammer IR is used to express most aspects of design-specific information, including timing constraints, floorplan information, power strap placement, Hammer IR is used for design flow information, like setting up file paths and tool installation directories; technology and tool plugin selection; and design-specific configurations, including timing constraints, floorplan information, power strap placement, design hierarchy, et cetera.

Hammer can take multiple Hammer IR files as inputs, which enables modularization and reuse. To facilitate this, Hammer IR includes a **meta** setting, which instructs the database engine on how to handle conflicting entries or allows it to post-process the input in some manner. Without a **meta**, subsequent definitions of a database key overwrite previous entries. The current **meta** settings and their uses are as follows.

- The **append** meta will append additional items to an existing list and store it in the specified key.
- The **crossappend** meta will cross-reference an existing list provided by the first element and append to it the list in its second, storing the result in the specified key.

[3]https://json.org
[4]https://yaml.org

- The **crossappendref** meta will cross-reference all elements and concatenate them, storing the result in the specified key.
- The **subst** meta will perform string substitution using other database elements on the provided string and store the result in the specified key.
- The **crossref** meta will cross-reference one key to another, effectively copying the contents of the key provided in the string.
- The **transclude** meta will read the file provided by the string and store it in the specified key.
- The **json2list** meta converts the key into a list.
- The **prependlocal** meta is used to prepend the local path of the config dictionary to the provided file path. This is useful for packaging IP, which allows the hammer IR to be specified using relative file paths without requiring complicated environment variable setups.
- The **deepsubst** meta is used to traverse a hierarchical set of keys and perform **subst**s on them. This also supports metas within the traversal.

The Hammer IR database is organized into namespaces hierarchically. For an example, refer to Listing 4.1. Hammer includes a default namespace for each tool, e.g. **syn**, **par**, **lvs**, **drc**, etc., and a global **vlsi** namespace for common settings among tools. Additionally, the user may create additional namespaces for custom hook usage and design settings. These are created automatically by Hammer if present in the Hammer IR file.

4.3.2 Tool plug-ins

Tool plug-ins are python packages that are dynamically loaded into the Hammer runtime based on the specified configuration in the Hammer database. Each plugin contains a **defaults.yml** file, which is a Hammer IR file responsible for setting up any default settings required by the plug-in. These are typically settings that most users will never change, which allows the user-specific Hammer IR to focus on design-specific changes. Any required fields are explicitly omitted from **defaults.yml** so that the Hammer driver will exit gracefully with an error message if absent. An example of the Vivado synthesis **defaults.yml** is shown in Listing 4.2, and an example of Python tool plug-in code is shown in Listing 4.3.

The Python code within the tool plug-in extends an abstract tool class specific to the tool type (e.g. **HammerSynthesisTool**, **HammerPlaceAndRouteTool**, etc.). Each tool has a specific set of abstract methods that the concrete class must implement, but tool plug-ins may include additional methods. The end result of this process is to produce a set of scripts and design collateral that the EDA tool uses to perform its task. Hammer also includes a separate method to execute the tool with the generated inputs. In normal operation, Hammer will perform these tasks serially, but Hammer's driver supports running specific tools in its invocation. This mode is used by the Makefile-based flow, which is the preferred method for implementing hierarchical designs.

The tool execution procedure is broken down into **steps**, which each corresponding to a specific code block within the generated tool script. Most often these scripts are written in Tcl[5], but there is nothing specific to Tcl within Hammer's code base. Each base tool type has its own set of steps which can be augmented by the plug-in using hooks. A hook is a custom step that is tagged to run before or after another step in the flow and is implemented as a Python function. Technology plug-ins and design-specific drivers may also add their own hooks.

4.3.3 Technology plug-ins

At a minimum, a technology plug-in contains a technology JSON file. This file specifies the locations of technology process development kit (PDK) files, such as the routing information, design rule decks, SPICE models for LVS, layer map files, and other such collateral. The technology JSON also contains useful technology-specific properties, like the optical shrink factor, metal stack-up choices, database units, and corner definitions. These are loaded into variables within the Hammer Python environment to be accessed by any built-in or custom methods within the flow. An excerpt from the technology JSON file for an open-source 7nm predictive technology, ASAP7[6] [101], is shown in Listing 4.4.

Technology plug-ins may optionally include a set of Python hooks to be injected into specific EDA tool flows. This is very useful for creating a technology platform that can be shared across multiple tapeouts, which drives reuse of solutions to common problems like design rule fixes. One example of such reuse from the Eagle tapeout was a hook that modified a specific standard cell to not be placed under a vertical metal 3 power rail. This particular cell had a metal 2 pin that could only be accessed by dropping a via down from metal 3, so placement under a metal 3 power strap creates an impossible routing task. Preventing this placement using a custom metal 3 blockage prevents the need to re-place portions of the design late in the cycle. This hook was reused by multiple tapeouts simultaneously. A particular challenge with this approach is that each technology-tool pair requires a unique set of hooks to accomplish a task. With proper abstractions, this interaction can be minimized, but for many new features that have yet to be adopted into core Hammer, this is unavoidable.

4.3.4 Build flow generation

For hierarchical flows, Hammer generates a Makefile in a format that can be easily included within a top-level project Makefile, which allows the build dependencies to be established based on the Hammer IR itself. This is useful for generators which may have different

[5]https://www.tcl.tk/

[6]https://github.com/The-OpenROAD-Project/asap7

optimal hierarchies based on the generator configuration. An example of this type of Makefile inclusion can be found in the Chipyard repository[7]. Listing 4.5 shows an excerpt of a generated Makefile from the EagleX chip, abridged for readability.

In this example, line 1 sets up the Hammer driver executable, and line 2 sets the list of prerequisite files based on the input configurations provided to the Hammer driver when generating the Makefile. Lines 8 through 15 establish global tasks which happen once per SoC. In this example, the only global task is to generate collateral for building a PCB and a package. Lines 19 through 23 show aliases for tasks for a given hierarchical cell (**Cluster**). The dependencies for these tasks are determined by the Hammer flow and the hierarchy. For a given hierarchy level, physical verification tasks like LVS and DRC are dependent on place-and-route, which is dependent on synthesis, which is dependent on place-and-route for any sub-hierarchical cells. An example of this dependency graph is included in Figure 4.1.

Lines 43 through 48 show a set of **redo-** targets, which intentionally bypass the Makefile dependencies to allow the user to rerun a task without invoking prior build rules. This feature allows the user to perform interactive tasks on the design and is especially useful while developing hooks, which can be tested easily by modifying local files and rerunning specific steps.

[7]https://github.com/ucb-bar/chipyard

```yaml
# Custom configuration keys used by hooks
eagle.num_lanes: 8
# Place-and-route keys
par.innovus.floorplan_mode: generate
par.innovus.power_straps_mode: generate
par.inputs.gds_merge: true
# Hierarchical flow keys
vlsi.inputs.hierarchical:
  mode: hierarchical
  top_module: EagleChipClockTopWithPads
  config_source: manual
  # Hierarchy specification
  manual_modules:
  - EagleChipClockTopWithPads:
    - Cluster
    - EagleTLLaneGroup
    - TLROM
    - RocketTile_20 # Systolic tile
    - RocketTile_21 # System control unit
  - Cluster:
    - RocketTile
  # Constraint specification
  constraints:
  - EagleTLLaneGroup:
    - vlsi.inputs.pin_mode: "generated"
    - vlsi.inputs.pin.assignments: [
      {pins: "*", layers: ["M4", "M6"], side: "right"},
      {pins: "tx_n tx_p rx_n rx_p VDDA", preplaced: true}
    ]
    - vlsi.inputs.clocks: [
      {name: "clock", period: "2ns", "uncertainty": "0.1ns"},
      {name: "ref_clock", period: "1.14ns", "uncertainty": "0.1ns"},
      {name: "tx_clock", period: "1.14ns", "uncertainty": "0.1ns",
        "path": "[get_pins pin:EagleTLLaneGroup/clock_tx_div]"},
      {name: "rx_clock", period: "1.14ns", "uncertainty": "0.1ns",
        "path": "[get_pins pin:EagleTLLaneGroup/clock_rx_div]"},
    ]
```

Listing 4.1: Example Hammer IR (YAML format) excerpt from EagleX.

```yaml
# Default settings for synthesis in Vivado
synthesis.vivado:
  # Location of the binary.
  binary: "vivado"
  # Location of the vivado setup script
  # type: Optional[str]
  setup_script: "vivado_setup.sh"
  # Location of the constraints file
  # type: Optional[str]
  constraints_file: "vc707.xdc"
  # FPGA board files to add to board repository.
  board_files: ""
  # FPGA board name
  board_name: "vc707"
  # FPGA part name
  part_fpga: "xc7vx485tffg1761-2"
  # FPGA board part
  part_board: "xilinx.com:vc707:part0:1.3"
  # DCP macro files directory.  Empty string for no DCP macro.
  dcp_macro_dir: ""
  # IP definition TCL file.  Empty string for no IP.
  ip_def_tcl: ""
  # Generate the TCL file but do not run it yet.
  generate_only: false
```

Listing 4.2: An example defaults.yml for Vivado synthesis.

```python
class VivadoSynth(HammerSynthesisTool, VivadoCommon):
  @property
  def steps(self) -> List[HammerToolStep]:
    return self.make_steps_from_methods([
      self.setup_workspace,
      self.generate_board_defs,
      self.generate_paths_and_src_defs,
      self.generate_project_defs,
      self.generate_prologue,
      self.generate_ip_defs,
      self.generate_messaging_params,
      self.generate_synth_cmds,
      self.run_synthesis,
    ])

  # Method to implement the run_synthesis step
  def run_synthesis(self) -> bool:
    syn_tcl_filename = os.path.join(self.run_dir, "syn.tcl")
    with open(syn_tcl_filename, "w") as f:
      f.write("\n".join(self.output))
    file_params = {
      'env_setup_script':
      self.get_setting('synthesis.vivado.setup_script'),
      'work_dir': self.run_dir,
      'vivado_cmd': self.get_setting('synthesis.vivado.binary'),
    }
    run_script = self.generate_run_script('run-synthesis', \
      file_params)
    # Run the executable
    self.run_executable([run_script])
    return True

  # Other step implementations and fill_outputs go here
  # ...

tool = VivadoSynth
```

Listing 4.3: An excerpt from the plug-in file for Vivado synthesis.

```
{
  "name": "ASAP7 Library",
  "grid_unit": "0.001",
  "time_unit": "1 ps",
  "tarballs": [
    {
      "path": "ASAP7_PDKandLIB.tar",
      "homepage": "http://asap.asu.edu/asap/",
      "base var": "technology.asap7.tarball_dir"
    }
  ],

  "gds map file": "ASAP7_PDKandLIB.tar/.../asap7_fromAPR.layermap",

  ...
}
```

Listing 4.4: An excerpt from the ASAP7 technology JSON file.

```
HAMMER_EXEC ?= eagle-hier-vlsi.py
HAMMER_DEPENDENCIES ?= eagle.yml eagle-pads-20T8H8M16BSysPMU.yml \
  eagle-pll.json eagle-sealring.json # ...

####################################################################
## Global steps
####################################################################
.PHONY: pcb
pcb: build/pcb-rundir/pcb-output-full.json

build/pcb-rundir/pcb-output-full.json: $(HAMMER_DEPENDENCIES)
  $(HAMMER_EXEC) -e bwrc-env.yml -p eagle.yml \
    -p eagle-pads-20T8H8M16BSysPMU.yml -p eagle-pll.json \
    --obj_dir build pcb

####################################################################
## Steps for Cluster
####################################################################
.PHONY: syn-Cluster par-Cluster drc-Cluster lvs-Cluster
syn-Cluster: build/syn-Cluster/syn-output-full.json
par-Cluster: build/par-Cluster/par-output-full.json
drc-Cluster: build/drc-Cluster/drc-output-full.json
lvs-Cluster: build/lvs-Cluster/lvs-output-full.json


build/syn-Cluster-input.json: build/par-Tile/par-output-full.json
  $(HAMMER_EXEC) -e bwrc-env.yml \
    -p build/par-Tile/par-output-full.json \
    -o build/syn-Cluster-input.json \
    --obj_dir build hier-par-to-syn

build/syn-Cluster/syn-output-full.json: build/syn-Cluster-input.json
  $(HAMMER_EXEC) -e bwrc-env.yml -p build/syn-Cluster-input.json \
    --obj_dir build syn-Cluster

build/par-Cluster-input.json: build/syn-Cluster/syn-output-full.json
  $(HAMMER_EXEC) -e bwrc-env.yml \
    -p build/syn-Cluster/syn-output-full.json \
    -o build/par-Cluster-input.json --obj_dir build syn-to-par

```

```
# ...

# Redo steps (for intentionally ignoring the dependency graph)
.PHONY: redo-syn-Cluster redo-par-Cluster

redo-syn-Cluster:
  $(HAMMER_EXEC) -e bwrc-env.yml -p build/syn-Cluster-input.json \
    $(HAMMER_REDO_ARGS) --obj_dir build syn-Cluster

# ...

####################################################################
## Steps for Top
####################################################################
syn-Top: build/syn-Top/syn-output-full.json
par-Top: build/par-Top/par-output-full.json
drc-Top: build/drc-Top/drc-output-full.json
lvs-Top: build/lvs-Top/lvs-output-full.json


build/syn-Top-input.json: build/par-Cluster/par-output-full.json \
  build/par-EagleTLLaneGroup/par-output-full.json # ...
  $(HAMMER_EXEC) -e bwrc-env.yml \
    -p build/par-Cluster/par-output-full.json \
    --obj_dir build hier-par-to-syn
```

Listing 4.5: An abridged Makefile for a hierarchical Hammer flow. Note that absolute paths have been trimmed to relative paths for readability, and many rules and prerequisites have been omitted, signified by an ellipsis.

4.4 Technology and EDA tool abstractions

Hammer provides IR abstractions for most concepts commonly used in digital ASIC flows, like timing constraints, placement constraints, obstructions, blockages, pin placements, bump placements, tap cell placement, power strap placement, et cetera. For the most part, these are thin layers that abstract away the tool-specific commands used to perform these tasks, as can be seen in Listing 4.1. However, physical design generators become more powerful when layers are built on top of these to automate the engineering tasks that determine their inputs. For such tasks, Hammer allows the user to select among the following options.

- The **empty** option will skip the given task.

- The **manual** option will use a manual specification provided by the user.
- The **generated** option will use a high-level generator API, with parameters provided by the user, to perform the task. These APIs include a **generate_mode** or **generate_method** key to specify which technique to apply, if multiple are available.

Power straps are a good candidate for this type of approach, as they intermix all three aspects of physical design: tool-specific commands, design-specific supply nets and low power strategy, and process-specific routing information.

4.4.1 Power straps

The power distribution strategy is critical for modern ASIC designs, as the interconnect scaling trends favor transistor density over wire resistance. As a result, proportionally more effort and routing resources must be devoted to the on-chip power grid to meet electrical reliability specifications like maximum IR drop. Compounded with the additional complexity of modern low-power flows driven by the power wall and the end of Dennard scaling, power grid design becomes a time-consuming engineering effort. However, specifying a power grid often requires a lot of fine-tuning to achieve optimal resistance while leaving sufficient routing resources for digital signals. The power grid in modern ASICs also must account for multiple voltage and power domains used by low power design flows.

Standard EDA tool commands require the user to specify widths, offsets, and pitches of power straps, which are often not easy to correlate with the routing tracks in the design. There may also be additional pitch constraints in the design due to bump or macrocell alignment requirements, as is the case in Eagle and EagleX.

Hierarchical cells which share a power domain must either have grids that connect exactly by abutting them, or they must build their power straps in intermediate metal layers and allow the top-level place-and-route job to create a top-level grid to connect them. The latter technique unnecessarily consumes intermediate metal routing resources which can be more efficiently utilized by clock and long-distance signal routing. However, abutment of hierarchical power grids creates complex interdependencies among all IPs, which must take into account their placement within the top level floorplan to align offsets correctly.

Hammer provides an API to generate power straps using a **by_tracks** method to streamline this process. Listing 4.6 contains an example power grid specification using this method for ASAP7. Rather than having the user specify specific sizing details, the **by_tracks** method, as its name suggests, allows the user to specify how many available routing tracks to devote to power routing as a percentage, along with offsets and widths of the straps in units of digital routing tracks. Because Hammer is aware of the metal design rules for each layer, this allows the Hammer tool to perform the required math to optimize the straps while allowing the engineer to reason about the design in a more intuitive and process-portable way. Figure 4.2 demonstrates this concept with a 50% power strap utilization and 4-track-wide straps. Figure 3.14 shows a multi-voltage top-level power strap plan used on the Eagle and EagleX chips. Figure 4.3 shows two different power strap specifications in ASAP7. The available options for the power strap **by_tracks** API are as follows.

- **strap_layers** specifies which layers shall contain power straps. Any layers that are omitted will only be used to create DRC-clean vias between other specified layers.
- **pin_layers** specifies which power strap layers shall contain pins, which is used by the hierarchical flow when creating physical abstractions like LEFs.
- **track_width** specifies the number of tracks to use for a single power strap of a single net. This setting may be appended with a layer name to specify the track width for that layer only.
- **track_spacing** specifies the number of tracks to leave between two straps of different nets within a power strap group. This is often zero. This setting may be appended with a layer name to specify the track spacing for that layer only.
- **track_start** specifies the first track on which to place a power strap, counting from the left for vertical layers and from the bottom for horizontal layers. This setting may be appended with a layer name to specify the track start for that layer only.
- **power_utilization** specifies the power utilization as a fraction (1.0 = 100%, i.e. no available signal routing tracks). This setting may be appended with a layer name to specify the power utilization for that layer only.

4.5 Future work

Hammer has been integrated into Chipyard and is consequently seeing rapid adoption, having reached over 100 Github stars and 200 unique cloners at the time of writing[8]. Support for the open-source Skywater 130nm PDK[9] is in development. Future work will involve the implementation and refinement of additional signoff flows, like power analysis, static timing analysis, and logical equivalency checking. While it is currently possible to run these flows on Hammer-based designs, they too benefit from the abstractions and flow automation offered by Hammer.

Timing and power analysis prediction [102] are interesting areas of research to integrate into Hammer. These, along with other types of metrics collection as described in [103], would allow closed-loop feedback to Chisel generators to enable autonomous design space exploration of large digital systems.

[8]https://github.com/ucb-bar/hammer/graphs/traffic
[9]https://github.com/google/skywater-pdk

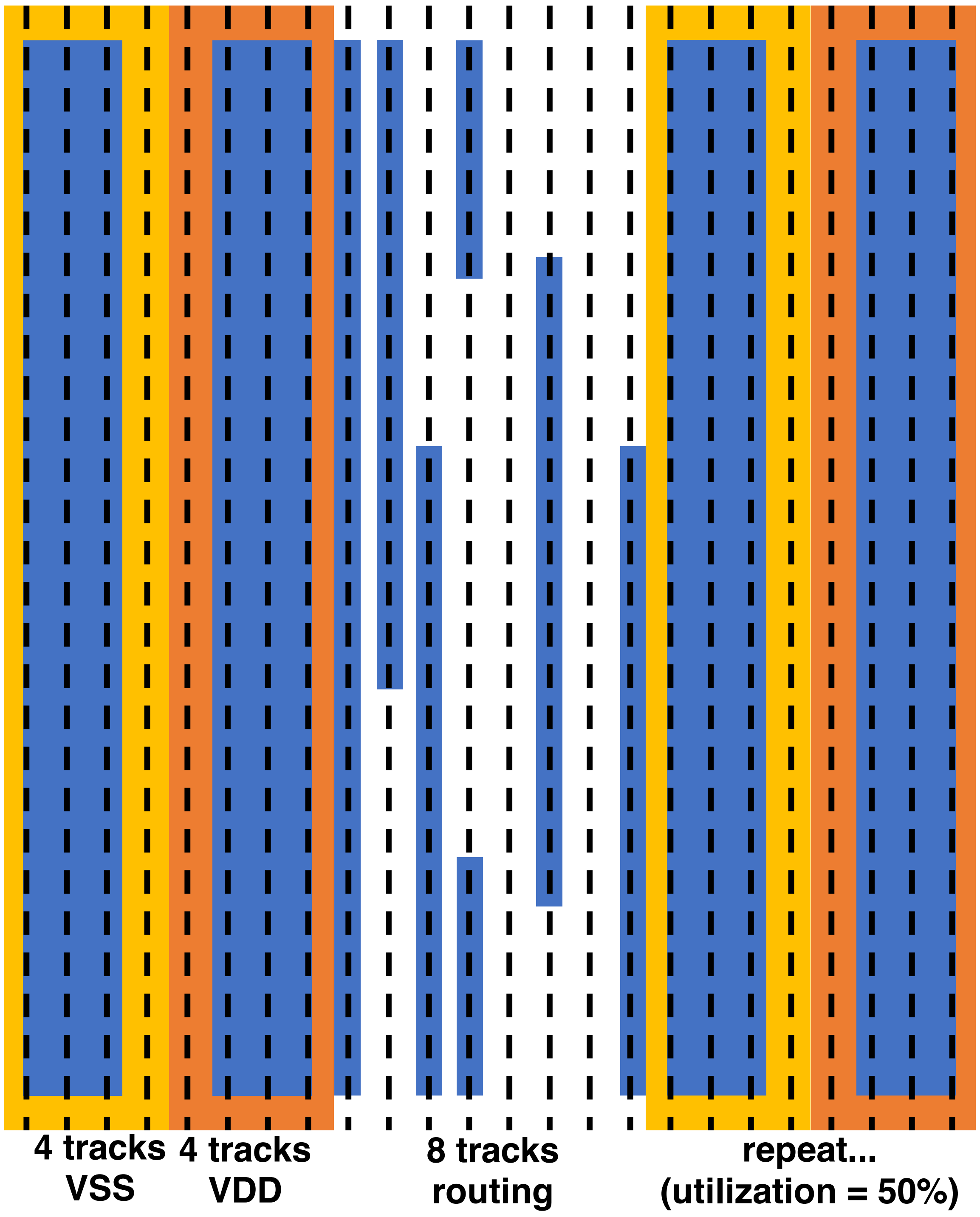


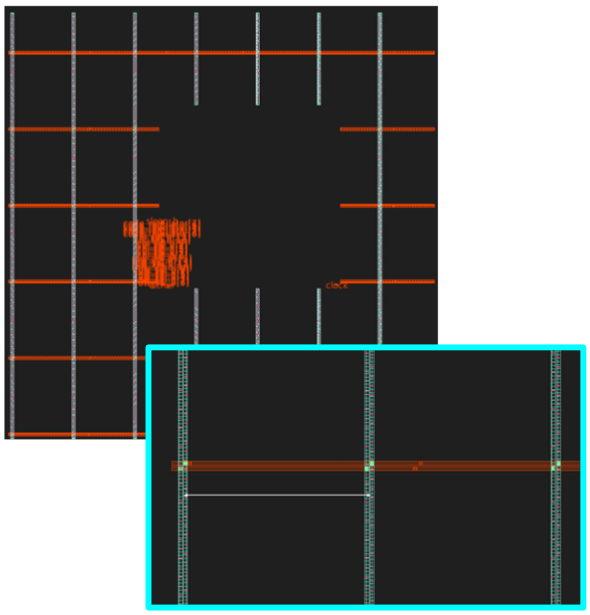
Figure 4.2: Simple power strap generation using the Hammer power strap generation API.

```
par.power_straps_mode: generate
par.generate_power_straps_method: by_tracks
par.generate_power_straps_options:
  by_tracks:
    strap_layers:
      - M3
      - M4
      - M5
      - M6
      - M7
      - M8
      - M9
    pin_layers:
      - M9
    track_width: 7
    track_spacing: 0
    track_spacing_M3: 28
    track_start: 10
    power_utilization: 0.25
    power_utilization_M3: 0.6
    power_utilization_M8: 1.0
    power_utilization_M9: 1.0
```

Listing 4.6: Hammer IR to specify power straps in ASAP7 using the by_tracks power strap generation API.

Power strap automation can be further improved by calculating the offsets of each hierarchical IP based on the top-level floorplan automatically, which is known at the time the flow is built.

The printed circuit board (PCB) support in Hammer is limited to simple schematic symbol and footprint outputs for a single PCB layout tool, Altium. The ramifications of chip-level decisions at the board-level are not addressed by this feature. There is interest in chip- and chiplet-package co-design [104] to ease board-level integration of generated systems-on-chip. Hammer is a good candidate for this type of co-design, and future research should consider incorporating this functionality.


(a) Sparse

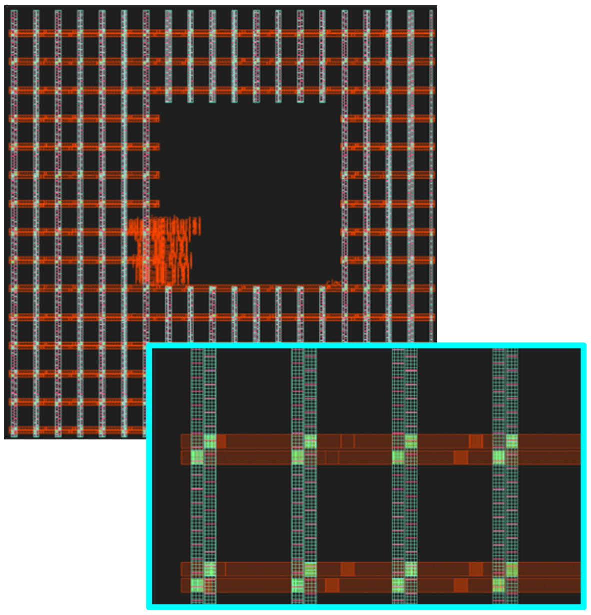

(b) Dense

Figure 4.3: An example of sparse and dense power straps using ASAP7.

Chapter 5

Floorplanning for Generated RTL

The physical design automation framework described in Chapter 4 requires an externally-supplied floorplan. While this framework is useful for developing EDA-tool-agnostic floorplan specifications, these floorplans remain specific to a particular design instance. This poses a problem for generator-based design flows, as relatively minor changes to the generator configuration may require labor-intensive changes to the floorplan. This issue can be mitigated by writing parameterized floorplan generators in a scripting language like Python, but this approach is fragile and is limited to simple, high-level parameterizations like the number of cores in a multicore CPU. Changes to the underlying generator or configurations beyond the scope of the floorplan generator would produce incorrect and unusable floorplans.

The difficulty with building a robust floorplan generator stems from the need for rich, deep knowledge of the parameterization space. It is possible to serialize the generator parameters into an external data structure that can be read by the floorplan generator, but this essentially requires the floorplan generator to recreate the contexts in which those parameters are used when determining placements and sizing. This also leads to the repetition of design intent, because the interpretation of the design parameters need to be described in both the RTL generator and the floorplan generator. This not only doubles the maintenance burden for a single generator, but it also creates a pathway for bugs to enter the design flow. These bugs are often not immediately obvious; it may take a multiple long-running place-and-route jobs, or ones with aggressive area or timing constraints, to highlight mistakes with the floorplan generator.

It is therefore preferable to design a floorplan by using as much source configuration data as possible. This chapter describes a novel floorplanning framework to enable floorplan generation for highly-parameterized Chisel designs. This framework aims to solve the problems associated with external floorplan generators by building the floorplan with data available at generator runtime, providing full access to the design configuration, type information, and logical hierarchy of the elaborated instance.

5.1 Architecture of a Chisel floorplanning framework

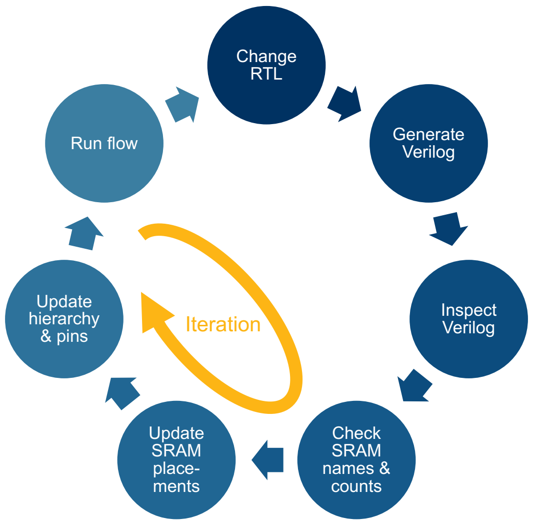


Figure 5.1: Chisel SRAM floorplanning flow without automation.

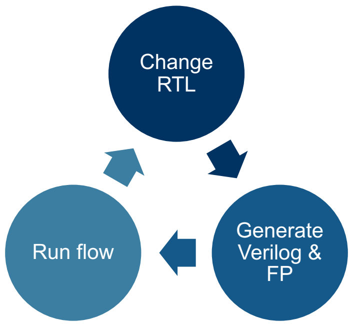


Figure 5.2: Chisel SRAM floorplanning flow with automation.

Although Chisel has been demonstrated to improve productivity for RTL generation, it is at the cost of increased floorplanning complexity. Figure 5.1 demonstrates the additional steps required when floorplanning Chisel-based designs. An ideal flow is shown in

Figure 5.2, which bypasses the labor-intensive steps of inspecting generated RTL. To close the gap between these flows, a floorplan generation framework designed specifically for this use is presented. To combat the challenges of constructing floorplans for generated RTL, an effective Chisel floorplan generation framework should adhere to the following tenets.

1. The framework must have access to the Chisel data structures and parameters.
2. The framework must tolerate renaming and hierarchical manipulation in FIRRTL.
3. The framework must support abstract Mem constructs used by Chisel generators.
4. The floorplan code must exist separately from the RTL.
5. Floorplan generators must be composable.
6. The final floorplan data must be emitted separately from the RTL.

Tenet 1 requires that the floorplan has access to the Chisel data structures and parameters so that the floorplan generator has access to the entire design without requiring serialization. This allows the floorplan generator to make conditional decisions on any design parameter without the overhead of explicitly tooling the RTL generator to emit the required information and avoids duplication of design information. To accomplish this, the floorplan application programming interface (API) is written in Scala so that it can execute in the same runtime environment as the Chisel engine. By using Scala, the floorplanning framework also benefits from its rich type system, improving type safety of the generator to prevent many classes of type-related programming bugs. Changes to the RTL which would prevent compilation also helps to keep the floorplan generator synchronized with the RTL generator, preventing another class of bug to which external floorplan generators are vulnerable.

Tenet 2 requires that the framework must tolerate renaming and hierarchical manipulation which occur naturally in the FIRRTL compiler. FIRRTL renames modules as they are de-duplicated or uniquified as requested. Additionally, SoC flows using FIRRTL frequently use passes to manipulate the hierarchy of the design, which allows for removal of extraneous levels of hierarchy, like test harnesses, or for alteration of hierarchical boundaries for physical design. Because these tasks happen within the FIRRTL compiler, they are transparent to the user at the Chisel level. It can be difficult to trace these events through the FIRRTL compiler to the generated output, especially when adding or removing FIRRTL passes to manipulate names and hierarchy. An external floorplan generation script would require specific knowledge of these transformations, and any updates to the sequence of FIRRTL passes used to make them would necessitate a parallel update to the floorplan generation script. However, a floorplanning API which interacts with the design within Chisel instead of the generated Verilog is able to ignore these transformations. To enable this, the floorplan API uses FIRRTL annotations to encapsulate the floorplan information. This floorplan information, called floorplan IR, or FPIR, contains geometric information about the desired layout of modules and macrocells within the design. FPIR is attached to design elements via

FIRRTL annotations, which maintain their relationship with their annotated nodes through renaming and hierarchy transformation. The resulting FPIR is emitted by the FIRRTL compiler using a custom FIRRTL pass, which executes once the standard FIRRTL passes have completed. At this point, the hierarchy transformation is complete, so the resulting FPIR file contains correct-by-construction paths to floorplanned design nodes. Floorplan IR is discussed in greater detail in Section 5.2.

Tenet 3 requires that the framework must support abstract **Mem** constructs, which Chisel uses to represent memories. In Verilog-based designs, users instantiate technology-specific SRAM macrocells directly, or by using wrapper modules. The modules which instantiate macrocells directly are therefore tied to a specific process technology. By using wrappers, designers can write RTL that is process-agnostic, but this requires that the designer create a conforming wrapper for each process technology in which the design is used. Chisel, on the other hand, has an abstract primitive type for memories, called a **Mem**, which represents multidimensional storage elements. The most common concrete class of **Mem** is **SyncReadMem**, which matches the semantics assumed by most SRAM designs, in that address and read enable are provided synchronously before the clock edge, and the corresponding read data updates after the clock edge. Write data and read enable are supplied along with address in the case of a write, and the write is committed after the clock edge. Because this is handled natively in Chisel, VLSI flows use a custom FIRRTL pass to replace **Mem** elements with groups of appropriately sized and connected SRAMs. This results in information created by the FIRRTL pass that is not known to the designer at the floorplan API level: the mapping of **Mem** constructs to SRAM groups.

To handle this, the floorplan API must be able to annotate chisel **Mem**s, and the floorplan framework back-end must be able to replace these specific annotations with special groups of macrocells, allowing the floorplan framework back-end to efficiently array the memories based on process-specific constraints. This replacement suggests a “Pass”-based architecture similar to FIRRTL, which is itself based on LLVM [105].

Tenet 4 requires that the floorplan generator code must exist separately from the RTL, meaning that the floorplan generator code should not need to exist (and should actually be discouraged from existing) in the same file or package as the RTL generator source code. This is important for multiple reasons. First, co-locating the RTL generator code and the floorplan generator code forces downstream users to use the floorplan generator implementation included with the RTL. This locks users out from exploring new floorplan ideas and prevents them from working around an issue with a floorplan generator that arises in their specific use case. It also pollutes the generator RTL code, obfuscating both logical intent and physical intent. Second, most systems-on-chip include some amount of third-party, or external, intellectual property (IP). It is undesirable, and sometimes restricted by license, to modify third-party IP to incorporate additional functionality like floorplan generation. Even in cases where all IP is internal, it may be challenging to re-release a frozen design to update the floorplan. Third, external floorplan code allows users to establish libraries of reusable floorplan components which can be interchanged with alternate implementations and published independently. This encourages a software-development-like ecosystem to improve the productivity of the entire community with access to the libraries.

However, it is difficult to implement such a scheme using object-oriented programming principles alone, as most object-oriented languages require the implementation for a single class to exist entirely within a single file. It is sometimes possible to circumvent this using inheritance, but generally it is difficult to re-inject the child class back into the context where it is used in the third party or external code. Aspect-oriented programming (AOP) [106] is a programming paradigm intended to solve this type of issue. Through the aspect construct, AOP allows users to add arbitrary code in a separate location that is later injected into specific points within the main code. There are many valid criticisms of AOP which make it unsuitable for most tasks, but the benefits for this particular case are encouraging. The Chisel language has a specific mechanism for supporting aspects [107]. This mechanism allows the user to run arbitrary code on one or more elaborated chisel modules by matching on their types. This is quite suitable for the floorplanning API as it guarantees type safety, provides full access to each instance of all floorplanned modules, and ensures that elaboration is complete, which guarantees that bit widths and port directions are finalized.

Tenet 5 requires that floorplan generators must be composable, meaning that multiple floorplan generators may be combined together to form a larger, more complex floorplan. This allows users to incorporate known-good floorplan generators from a floorplan generator library into their own floorplan generators and modularize complex floorplans by referencing sub-designs. This also enables the incorporation of hand-written floorplans into the generator to allow users to fine-tune portions of the floorplan or reuse existing arrangements without losing the benefits of generators. Chisel's AOP implementation also helps with this requirement. The Chisel AOP mechanism uses a partial function to execute code on modules for which that partial function is defined. By matching on type, aspects can be executed on specific modules of that type. In scala, partial functions are composable, so the floorplan API should encourage users to implement their floorplans using partial functions, which can be composed within the Chisel aspect to produce the top-level floorplan.

Tenet 6 requires that the final floorplan data be emitted separately from the RTL. Because large Chisel design have long run times, it is preferable to iterate on the floorplan, when possible, without rebuilding Chisel. It is also important to be able to modify floorplan parameters without touching the floorplan generator code, which requires the same run-time environment as Chisel. This allows the user to produce multiple floorplans for design space exploration or to annotate the floorplan with concrete design information like area, which changes in each process node. This requirement leads to a standalone floorplan compiler executable, which executes the "Passes" described previously. The Chisel API produces the highest abstraction level FPIR, which is lowered by passes until the final concrete floorplan is produced. Among these passes is a out-of-band annotation pass, which modifies the FPIR with instance-specific information like area or length constraints. The floorplan compiler can either emit concrete FPIR, with fixed sizes and placements, or hammerIR to be consumed directly by Hammer.

5.2 Floorplan IR

The Chisel floorplan compiler utilizes an intermediate representation called floorplan IR, or FPIR, to store floorplan information. Components of FPIR are called **Element**s. During Chisel elaboration, FPIR elements are attached to FIRRTL nodes via FIRRTL annotations, which allows them to persist through renaming and hierarchy manipulation. Elements attached to nodes that are removed by FIRRTL are also deleted; this may happen if the parent circuit is changed manually or modules are deleted by dead code elimination. There are four classes of floorplan annotations in the floorplan compiler framework:

- **NoReferenceFloorplanAnnotation** is used to annotate floorplan elements that are not tied to specific hardware, like spacers.
- **InstanceFloorplanAnnotation** is used to annotate floorplan elements that describe a module instance.
- **MemFloorplanAnnotation** is used to annotate **Mem** references.
- **FloorplanIRFileAnnotation** is used to specify the output FPIR JSON file.

NoReferenceFloorplanAnnotation extends the FIRRTL **SingleTargetAnnotation** class, targeting a single **InstanceTarget**. This target is called the scope, and refers to the scope of the current floorplan context. Scopes are discussed more in Section 5.4. Both **InstanceFloorplanAnnotation** and **MemFloorplanAnnotation** extend the FIRRTL **MultiTargetAnnotation** class, which allows a single annotation to have multiple targets. In both cases, only two targets are allowed. The first target in the targets list is the scope and the second target is the individual component described by the floorplan element. For **InstanceFloorplanAnnotation**s, this component must be a module instance. For **Mem-FloorplanAnnotation**s, this component must be a reference to a **Mem** node. While both are two-element **MultiTargetAnnotation**s, the underlying implementation of **MultiTargetAnnotation** does not preserve the typing of the underlying **Target**s, so separate annotation types are used to preserve type safety within the FIRRTL pass.

These annotations are consumed by a custom FIRRTL pass, **GenerateFloorplanIR-Pass**, which collects all remaining floorplan-related annotations after FIRRTL has emitted Verilog and from them constructs a **FloorplanState** object. A **FloorplanState** is the top-level container for serializing floorplan IR and contains a sequence of **FloorplanRecord**s and a level identifier. The level identifier is the maximum level of all elements within the state–an element's level refers to its level of abstraction. Levels range from 4, the most abstract level, to 0, a fully concretized floorplan. A **FloorplanRecord** is a database entry that records the floorplan element information, the scope, and the instance path and base module, if applicable. By separating the floorplan element information from the hierarchical information in the scope, instance path, and base module, this system is able to cleanly separate the geometry-related information from the hierarchical information, which prevents the Chisel

API from needing to handle hierarchy. The created **FloorplanState** object is serialized to a JSON string and written to the file specified by the **FloorplanIRFileAnnotation**. An example of this file format with a single entry is shown in Listing 5.1.

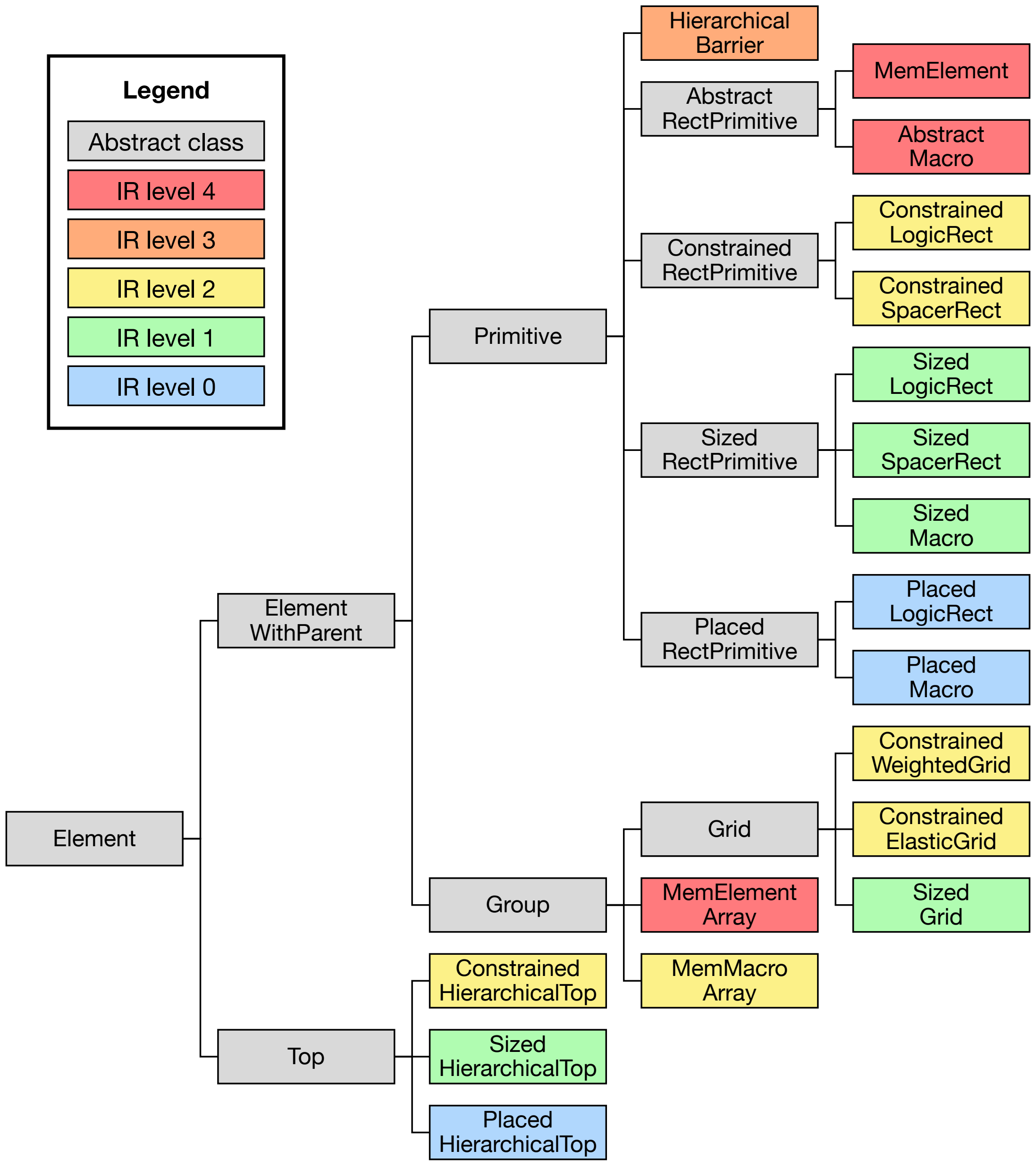


Figure 5.3: Floorplan IR class hierarchy and levels.

```
{
  "records":[
    {
      "scope":"ChipTop",
      "ofModule":"ChipTop",
      "element":{
        "class":"barstools.floorplan.ConstrainedHierarchicalTop",
        "name":"ChipTop_0",
        "topGroup":"unnamed_0",
        "width":{
          "class":"barstools.floorplan.Constrained",
          "eq":500.0
        },
        "height":{
          "class":"barstools.floorplan.Constrained",
          "eq":500.0
        },
        "area":{
          "class":"barstools.floorplan.Unconstrained"
        },
        "aspectRatio":{
          "class":"barstools.floorplan.Unconstrained"
        },
        "margins":{
          "left":0,
          "right":0,
          "top":0,
          "bottom":0
        },
        "hardBoundary":true
      }
    }
  ],
  "level":4
}
```

Listing 5.1: Serialized floorplan IR example.

5.2.1 Floorplan IR design

Figure 5.3 illustrates the class hierarchy of floorplan elements. The abstract class hierarchy allows logical grouping of similar elements and reduces repeated code. Additionally, some inter-element relationships imply specific abstract types for the relatives, but this is unenforceable by the type system, so the Chisel API must safeguard against violating types. The descriptions for the abstract classes are as follows.

- **Element** is the common ancestor of all floorplan IR elements.
- **Top** represents the top-level of a scope.
- **ElementWithParent** is not the top-level of a scope and has a parent element.
- **Group** contains a group of one or more sub-elements.
- **Grid** specifies a two-dimensional grid arrangement of sub-elements.
- **Primitive** is neither a **Top** nor a **Group** and therefore has no children.
- **AbstractRectPrimitive** is a rectangular primitive that cannot have constraints.
- **ConstrainedRectPrimitive** is a rectangular primitive that can be constrained.
- **SizedRectPrimitive** is a rectangular primitive with a concrete width and height.
- **PlacedRectPrimitive**: is a **SizedRectPrimitive** with a fixed location.

The concrete floorplan IR elements are categorized into one of five levels. Each level corresponds to a specific level of abstraction, with higher levels allowing more abstract components. These levels are listed below.

Level 4: Abstract memories and macrocells may exist without any physical dimensions.

Level 3: All memories must be mapped to macrocell instances, and all macrocells must have physical dimensions annotated via out-of-band annotations. Multiple ***HierarchicalTop** elements are allowed as long as all but the topmost have a corresponding **HierarchicalBarrier**.

Level 2: Only one ***HierarchicalTop** may exist, meaning that the entire floorplan state is within a single scope. No **HierarchicalBarrier**s may exist.

Level 1: All elements must have concrete sizes (i.e. no inequality constraints), but can still exist within sized groups.

Level 0: All elements must have concrete sizes and fixed locations. Only **Placed*** elements are allowed.

5.2.2 Floorplan IR element descriptions

Some floorplan elements are able to take constraint parameters for width, height, area, or aspect ratio. Floorplan IR includes a serializable constraints class hierarchy to specify these. It includes **Unconstrained**, which is always satisfied, **Impossible**, which is never satisfied, and **Constrained**, which has fields **eq** (equal to), **leq** (less than or equal to), **geq** (greater than or equal to), and **mof** (multiple of). Each of these fields can be optionally set to specify the constraint value.

Floorplan elements all have a name identifier that is unique to the scope of the floorplan context. The Chisel API enforces this, but any manually generated FPIR must also guarantee this property. These identifiers are used to link **Group** elements to their children and vice versa. Because the final hierarchical path is not known at elaboration time, the name is the only handle available to the floorplan IR elements for describing relationships. The FPIR for all **Group** elements contains a list of strings for the sub-elements. The semantic meaning of the ordering of this list is class-specific. When two floorplan scopes are combined, the child scope converts all names into new unique names within the parent scope. Therefore, names cannot be guaranteed to remain static throughout a floorplan compilation, but the names will be guaranteed to remain coherent. The full list of concrete floorplan elements and their uses is listed below.

Level 4

- **MemElement** describes a single **Mem** node that will be replaced with zero or more SRAM instances by the FIRRTL compiler.
- **AbstractMacro** describes a macrocell with unknown sizing. All **AbstractMacro**s must be sized using out-of-band annotations or the floorplan compiler will error.
- **MemElementArray** is a **Group** container for one or more **MemElement**s that should be treated as a single bank of memories during floorplan compilation. This is converted to a **MemMacroArray** once **MemElement**s are replaced with **AbstractMacro**s, which signifies to the floorplan compiler that contained macrocells are SRAMs and should be placed accordingly.

Level 3

- **HierarchicalBarrier** is used to reference a sub-floorplan in a different scope which is annotated with a ***HierarchicalTop** element. The instance paths for both elements must be the same (although the scopes will necessarily be different). This design decision enables floorplans to be composable by not requiring a single floorplanning context for the entire floorplan. This also allows the user to instantiate a hand-written floorplan within the generator for fine tuning or design reuse.

Level 2

- **ConstrainedHierarchicalTop** describes the constraints on a top-level floorplan, including dimensions and margins. A **ConstrainedHierarchicalTop** must have a single child element, **groupTop**, that is a **Group** type. The **groupTop** covers the entire floorplan at the scope of the described top module.
- **MemMacroArray** describes an array of SRAM macrocells that have no predetermined arrangement but may have width, height, area, or aspect ratio constraints.
- **ConstrainedLogicRect** describes a logic module instance that contains no floorplanned sub-components but must be encapsulated within a rectangle with the provided constraints.
- **ConstrainedSpacerRect** describes a filler element that is not attached to any hardware. This element is used to provide spacing between floorplan components, but is pruned before floorplan emission. It does not block un-floorplanned standard cell placement or routing.
- **ConstrainedWeightedGrid** describes a grid arrangement of floorplan elements where all rows are related to each other by given row weight constraints and all columns are related to each other by given column weight constraints.
- **ConstrainedElasticGrid** describes a grid arrangement of floorplan elements where all elements of a given row have the same height and all elements of a given column have the same width, but there are no specified relationships among rows or columns.

Level 1

- **SizedLogicRect** describes a logic module instance that has no sub-components and a fixed width and height.
- **SizedSpacerRect** describes a filler element that is not attached to any hardware and has a fixed width and height. This element is used to provide spacing between floorplan components, but is pruned before floorplan emission. It does not block un-floorplanned standard cell placement or routing.
- **SizedMacro** describes a macrocell instance that has a fixed width and height.
- **SizedGrid** describes a grid arrangement of floorplan elements where all rows and columns have specified dimensions.
- **SizedHierarchicalTop** describes the top-level floorplan dimensions and margins.

Level 0

- **PlacedLogicRect** describes a logic module instance that has no sub-components and a fixed width, height, and placement location.

- **PlacedMacro** describes a macrocell instance that has a fixed width, height, placement location, and orientation.
- **PlacedHierarchicalTop** describes the top-level floorplan offset, dimensions, and margins.

5.3 Floorplan compiler

The floorplan compiler consumes one or more floorplan IR files, memory instance map files, and out-of-band annotation files to produce a floorplan output, which can either be level 0 FPIR or hammerIR. This flow is shown at a high level in Figure 5.4. The primary mechanism for generating floorplan IR files is the Chisel floorplan API, but it is possible to generate them by hand or through a script. An example of the contents of a floorplan IR file is shown in Listing 5.1.

The floorplan compiler expects the memory instance map file to contain one line for each **MemElement**. Each line begins with the module name of the **MemElement** that has been replaced, followed by a whitespace-delimited list of macrocell instances. Each macrocell instance contains the new instance name for the specific macrocell and its module name delimited by a colon. Listing 5.2 illustrates an example memory instance map file.

Out-of-band annotations use the JSON format and describe modules rather than instances, which is better suited for this type of annotation. An out-of-band annotation can define any combination of the width, height, or area of a module. Currently only equality constraints are supported. Any module can be annotated, but **AbstractMacro**s require that both width and height are specified, which is the mechanism by which the floorplan compiler discerns their size. Generally this information is extracted from a physical view of the macrocell, like a LEF file, but heuristic technique like a FIRRTL pass that approximates area [102] would also be appropriate. Listing 5.3 shows an example of an out-of-band annotation file.

```
tag_array_ext mem_0_0:SRAM1RW64x22 mem_0_1:SRAM1RW64x22
tag_array_0_ext mem_0_0:SRAM1RW64x21 mem_0_1:SRAM1RW64x21
data_arrays_0_0_ext mem_0_0:SRAM1RW512x32 mem_0_1:SRAM1RW512x32
l2_tlb_ram_ext mem_0_0:SRAM1RW1024x44
```

Listing 5.2: Example of a floorplan compiler memory instance map file.

```json
[
    {
        "height": 180.46,
        "ofModule": "SRAM1RW1024x17",
        "width": 79.568
    },
    {
        "height": 300.94,
        "ofModule": "SRAM1RW1024x64",
        "width": 173.728
    },
    {
        "height": 75.38,
        "ofModule": "SRAM1RW512x8",
        "width": 47.008
    },
    {
        "height": 56.528,
        "ofModule": "SRAM1RW64x22",
        "width": 24.128
    },
    {
        "height": 56.556,
        "ofModule": "SRAM1RW64x21",
        "width": 23.248
    },
    {
        "height": 150.568,
        "ofModule": "SRAM1RW512x32",
        "width": 89.248
    },
    {
        "height": 301.048,
        "ofModule": "SRAM1RW1024x44",
        "width": 120.928
    }
]
```

Listing 5.3: Example of an out-of-band annotation file setting memory dimensions.

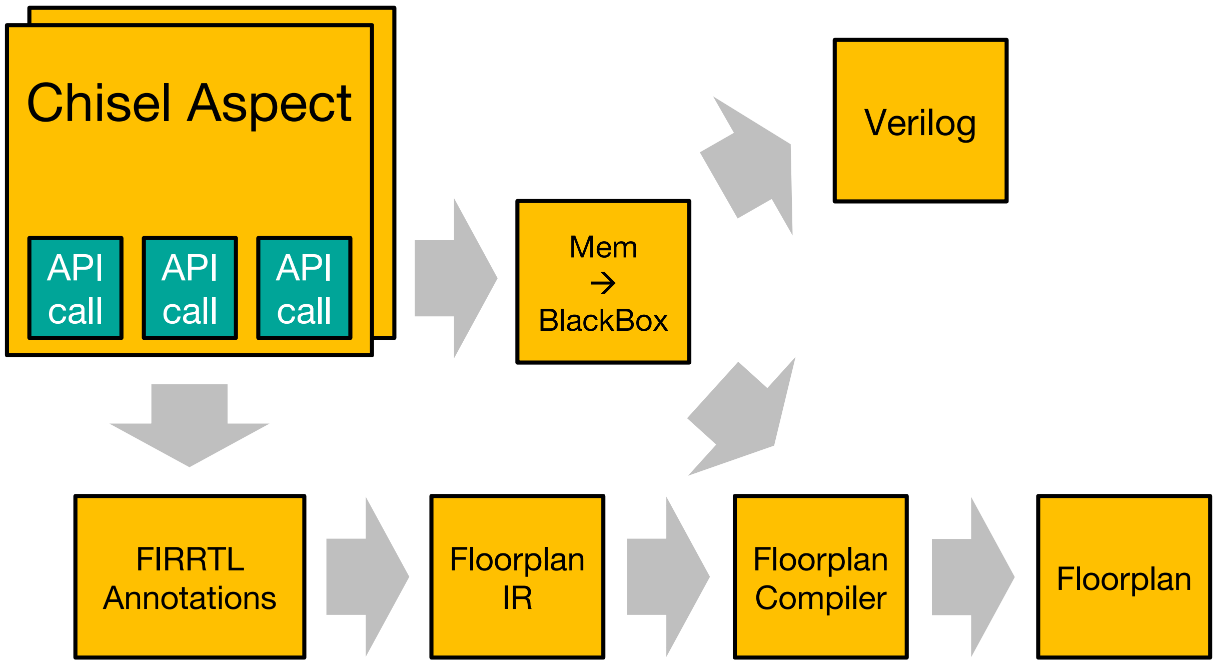


Figure 5.4: Chisel floorplan compiler flow.

5.3.1 Floorplan compiler passes

The floorplan compiler transforms floorplan IR by permuting a **FloorplanState** object through a series of passes as illustrated in Figure 5.5. Floorplan compiler passes must emit a **FloorplanState** that is the same level or lower than its input state. These passes transform **FloorplanRecord**s by modifying or changing their elements as described in the list of passes below. Most transformations preserve the instance and module information in the **FloorplanRecord**, but some, namely **ReplaceHierarchicalPass**, modify the scope and relative instance path of the records while preserving the absolute instance path.

- **TransformMemsPass** converts all **MemElement**s in the state into **Abstract-Macro**s. A single **MemElement** may be transformed into zero, one, or multiple **AbstractMacro**s, depending on the mapping provided in the memory instance map file. A **MemElement** which is mapped to a flip-flop-based memory is simply removed from the design, while its parent container remains. This pass also converts **MemElementArray**s into **MemMacroArray**s, while preserving the one-to-many parent-child relationship each has with the newly transformed **AbstractMacro**s.

- **OutOfBandAnnotationPass** adds additional constraints to **FloorplanRecord**s that match module names provided in the out-of-band annotation file. This is a mandatory step for any **AbstractMacro** elements in the design.

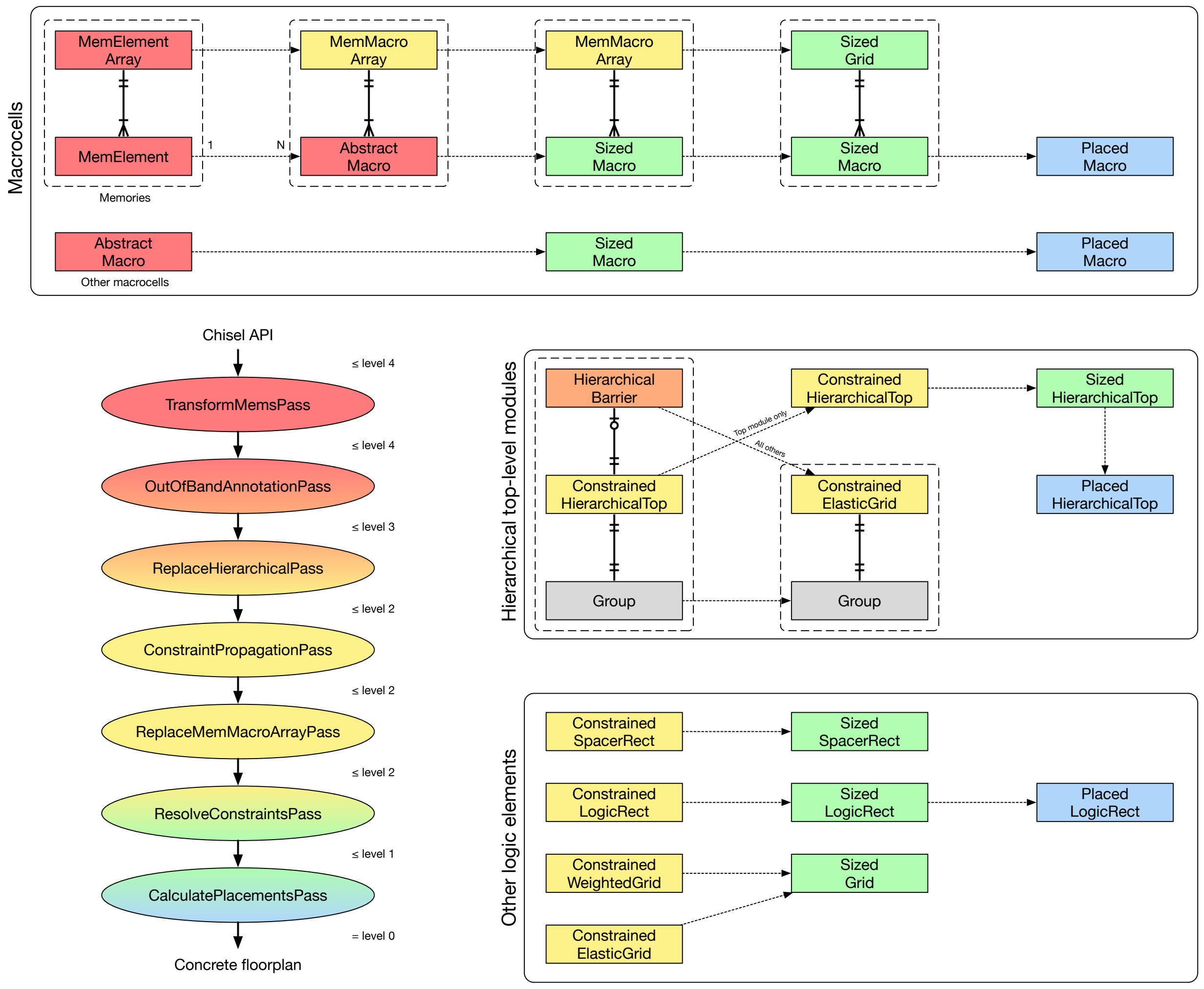


Figure 5.5: Floorplan compiler passes and FPIR lowering flows. All dotted lines represent 1:1 transformations unless otherwise noted.

- **ReplaceHierarchicalPass** consolidates all scopes in the design into a single floorplan scope. A scope will always have a **ConstrainedHierarchicalTop**, **SizedHierarchicalTop**, or **PlacedHierarchicalTop** at its root. For modules other than the topmost module, this must correspond to a **HierarchicalBarrier** at a higher level in the hierarchy. Each **ConstrainedHierarchicalTop** or **SizedHierarchicalTop** contains a single child called the **topGroup**, which is a **Group** element that represents the layout of the entire module. This pass will stitch the two disjoint scopes together by replacing the **HierarchicalBarrier** with a single-row, single-column **ConstrainedElasticGrid** containing the **topGroup**. After this replacement, the scope of each element from the child floorplan is replaced with the new scope and the element names are uniquified.
- **ConstraintPropagationPass** propagates constraints up and down the floorplan hi-

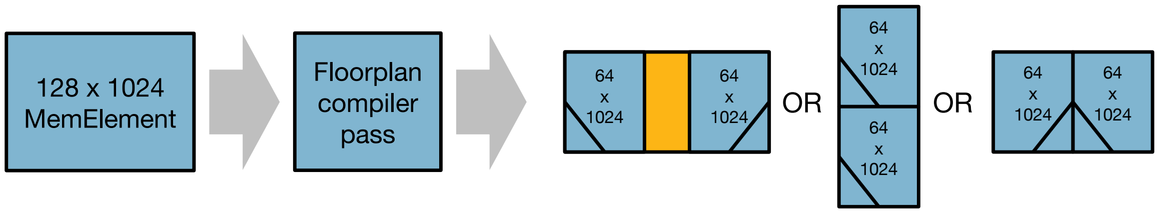


Figure 5.6: SRAM mapping using the floorplan compiler.

erarchy and to sibling elements of groups, if applicable. Any constraints which cannot be met are replaced with **Impossible** constraint objects.

- **ReplaceMemMacroArrayPass** transforms **MemMacroArray** groups into **SizedGrid** groups that describe the concrete layout of the memory array using the available constraints to determine topology. Figure 5.6 illustrates this process.
- **ResolveConstraintsPass** concretizes all constraints, optimizing for minimum area when resolving inequalities.
- **CalculatePlacementsPass** replaces all **Sized*** elements with **Placed*** elements by calculating absolute position within the floorplan. All **Group** and **SizedSpacerRect** elements are removed after they are used to determine positioning.

5.3.2 SRAM replacement

Placing SRAMs is a top priority when building a floorplan. In most modern process technologies SRAM macrocells have signal pins on only one of its four sides. When floorplanning, these pins must be unobstructed to be routable by the place-and-route tool. This typically means the SRAM macrocells can be abutted along any of the three non-pin-bearing sides, but not the fourth with pins. Depending on the implementation of the SRAM macrocell, this abutment can either be direct, intentionally shorting the power and ground rails, or require a small offset to meet design rules. Typically direct abutment allows for the most compact layout, as many gaps in the layer geometries, like N-wells, can be eliminated.

Figure 5.6 demonstrates how the floorplan compiler can arrange a small set of memories using these constraints. Figure 5.7 graphically illustrates the decision flow performed within **ReplaceMemMacroArrayPass** to arrange these SRAMs inside a constrained rectangular boundary.

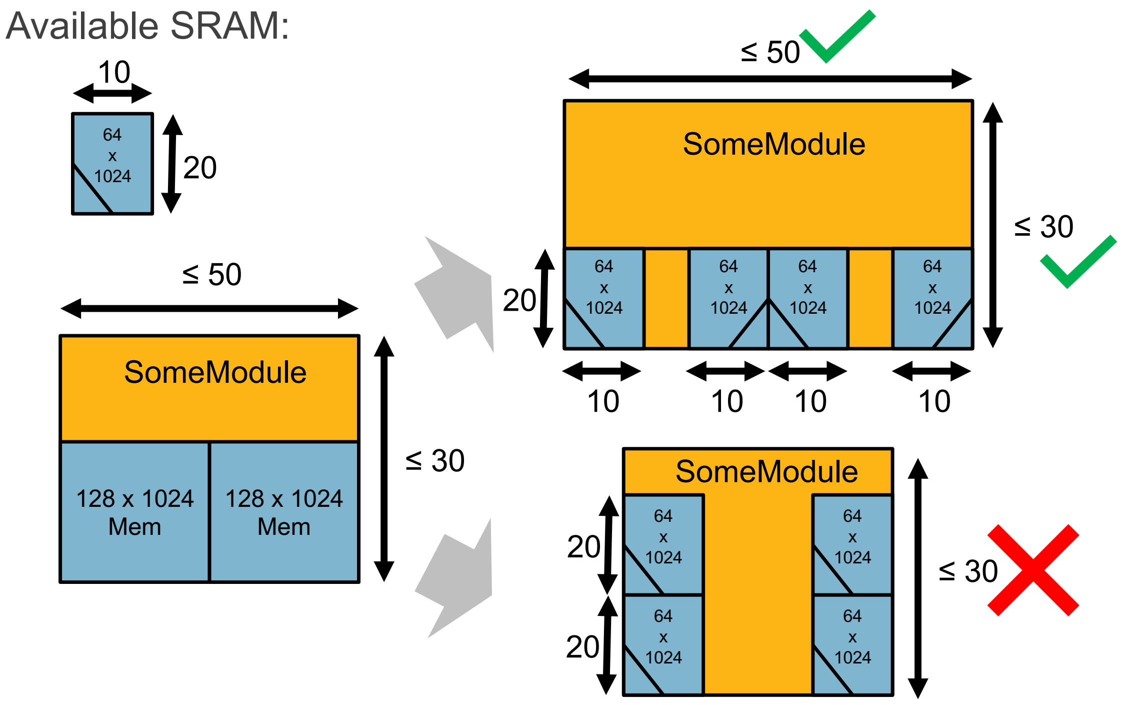


Figure 5.7: SRAM legalization flow using the floorplan compiler.

5.4 Chisel floorplan API

A Chisel floorplan generator is constructed using a Chisel **Aspect**. In Chisel, an **Aspect** is used to execute arbitrary code after a module of a specified type has been elaborated. This allows the addition of arbitrary additional hardware or the execution of code that inspects the design to perform a specific action.

Floorplans are created by providing a **FloorplanFunction** partial function to a **FloorplanAspect**. An **FloorplanFunction** is a partial function which takes a single Chisel **BaseModule** parameter and returns a **Seq[ChiselElement]**, which is a list of Chisel floorplan API components. The **FloorplanFunction** creates a **FloorplanContext** object, which provides an API to create and modify floorplan information within the given module instance, which becomes the scope of all floorplan components. These components are returned by calling the **commit** method of the **FloorplanContext** object, which ensures that any mutable data structures, like grids, are resolved legally by inserting spacer cells where required. The **FloorplanAspect** searches for modules for whose type the partial function is defined and calls it on each. The resulting **ChiselElement**s are collected and converted into serialized floorplan IR, which is annotated to the design using one of the floorplan annotations discussed in Section 5.2. Multiple partial functions can be composed to combine floorplan generators from different packages using the **orElse** operator.

Within a **FloorplanFunction**, the user creates a floorplan context by calling the **Floorplan** apply method. This context sets the scope of the current floorplan to the provided module, which is typically the input module to the function, and eventually creates a **ConstrainedHierarchicalTop** when FPIR is emitted. All floorplan elements are created through this floorplan context, or a child **ChiselGroup** created by the context, which prevents the user from needing to maintain scope information. Any elements created within the context are added to its **elements** list, which must be returned at the end of the floorplan function. A simple example of this is shown in Listing 5.4.

All elements returned by this are objects are descendants of the **ChiselElement** class. Once all floorplan functions have been executed, the concatenated sequence of **ChiselElement**s is converted to a sequence of FIRRTL annotations containing the serialized FPIR.

Most FPIR **Element** types have analogous **ChiselElement** classes with a one-to-one correspondence. For example, a **ConstrainedLogicRect** is created by a **ChiselLogicRect** class within the Chisel API. This is also true for the class hierarchy, with **ChiselGroup** mirroring **ChiselElement**, et cetera. The Chisel API only emits the highest level of a particular element type, meaning there is no mechanism to generate a **SizedLogicRect** FPIR element. However, using **EqualTo** constraints allows the user to achieve equivalent behavior without needing to add additional type support to the Chisel API. Chisel elements are created by calling **create** methods on the floorplan context object, which applies the correct scope to the element upon creation. To insert elements correctly into the hierarchy, **ChiselGroup** classes provide **placeAt** methods, which places the given element at the specified index within the group and creates the appropriate parent-child relationship necessary to produce valid FPIR. The semantic meaning of the index is dependent on the concrete **ChiselGroup** class.

Memories are treated differently than other floorplan components. Because a **MemElement** must be placed in a **MemElementArray**, no method exists within the floorplan context to add a Chisel **Mem** directly. Instead, the **MemElementArray** class acts as a mini-context, allowing the user to call **addMem** to create and attach the appropriate element objects.

The Chisel API allows the user to provide meaningful names for all floorplan elements, but this is optional. These names are treated as suggestions to avoid conflicts, and a singleton name array is maintained at runtime to guarantee uniqueness by appending suffixes to the suggestions if necessary.

```
object ExampleFloorplans {

  def apply(): FloorplanFunction = {
    // floorplan for module named "Top"
    case top: Top =>
      val context = Floorplan(top, 100.0, 100.0)
      val array = context.createElasticArray(2)
      val topGrid = context.setTopGroup(array)
      // Assign instance A to a rectangular area
      topGrid.placeAt(1, context.createRect(top.instA))
      // Cross-reference a separately defined floorplan of B
      topGrid.placeAt(0, context.addHier(instB))
      // return all elements from context
      context.commit()
    // floorplan for module named "Bar"
    case bar: Bar =>
      val context = Floorplan(bar)
      val array = context.createElasticArray(3)
      val topGrid = context.setTopGroup(array)
      topGrid.placeAt(2, context.createRect(top.instC))
      topGrid.placeAt(1, context.createSpacer())
      topGrid.placeAt(0, context.createRect(instD))
      // return all elements from context
      context.commit()
  }

}

case object ExampleFloorplanAspect extends FloorplanAspect[Top](
  ExampleFloorplan()
)
```

Listing 5.4: A simple example floorplan generator.

5.5 Example floorplans

Listing 5.5 and Listing 5.6 show floorplan generators for modules included in a default Chipyard build, **RocketTile** and **ChipTop**, respectively[1]. An instance of this floorplan generated for a custom **ChipTop RocketConfig** instance with no L2 cache is shown in Figure 5.8. This configuration includes a single rocket core with separate 16 KiB instruction and data caches. This core is not configured with an instance of the Hwacha vector accelerator, but the code shown includes a simple example to demonstrate the flexibility and composability of the approach.

In Listing 5.5, lines 4 through 9 import the classes and objects from the source RTL generators and the floorplan framework. Lines 14 through 58 define the floorplan generator for a **RocketTileModuleImp** instance, which is a Rocket CPU tile, and lines 59 through 64 define the floorplan generator for **HwachaImp**, a Hwcha instance. For the Rocket tile instance, line 15 creates the floorplan context. Next a 3-element vertical array is created, which will contain any accelerators in the top cell, a spacer in the middle cell, and the caches in the bottom cell. Line 18 places a 5-element horizontal array in the bottom cell, which is subsequently filled with the data and tag arrays for the L1 instruction and data caches separated by a spacer. Lines 30 through 34 of Listing 5.6 show the composition of the **ChipTop** floorplan and **RocketTile** floorplan, which are written in separate files.

[1]Because Chipyard uses the Diplomacy [108] library, the floorplans annotate the lazy module implementations rather than the lazy modules themselves, which is why the **ChipTop** floorplan targets the **ChipTopLazyModuleImp** module, which is the class that extends the Chisel hardware module class.

```scala
// See LICENSE for license details
package chipyard.floorplan

import freechips.rocketchip.tile.{RocketTileModuleImp}
import freechips.rocketchip.rocket.{HasHellaCache, DCache}
import hwacha.{Hwacha, HwachaImp}
import barstools.floorplan.chisel.{FloorplanAspect, Floorplan}
import barstools.floorplan.chisel.{FloorplanFunction, Direction}
import barstools.floorplan.{GreaterThanOrEqualTo}

object RocketFloorplans {

  def default: FloorplanFunction = {
    case tile: RocketTileModuleImp =>
      val context = Floorplan(tile)
      val topArray = context.createElasticArray(3)
      val topGroup = context.setTopGroup(topArray)
      val cacheArray = topGroup.placeAt(0,
        context.createElasticArray(5, Direction.Horizontal))

      // Place I£ data
      val icacheData = cacheArray.placeAt(0,
        context.createMemArray(Some("l1_icache_data")))
      tile.outer.frontend.icache.module.data_arrays.map { x =>
        icacheData.addMem(x._1) }
      // Place I£ tags
      val icacheTags = cacheArray.placeAt(1,
        context.createMemArray(Some("l1_icache_tags")))
      icacheTags.addMem(tile.outer.frontend.icache.module.tag_array)

      cacheArray.placeAt(2, context.createSpacer(
        name = Some("cache_spacer"),
        width = GreaterThanOrEqualTo(500)))

      tile.outer match {
        case x: HasHellaCache =>
          val dcacheData = cacheArray.placeAt(4,
            context.createMemArray(Some("l1_dcache_data")))
          val dcacheTags = cacheArray.placeAt(3,
            context.createMemArray(Some("l1_dcache_tags")))
          x.dcache match {
            case cache: DCache =>
              cache.module.dcacheImpl.data.data_arrays.map { x =>
                dcacheData.addMem(x._1) }
              dcacheTags.addMem(cache.module.dcacheImpl.tag_array)
            case _ =>
              ???
          }
        case _ =>
          // Do nothing
      }

      // Add optional accelerator placements
      val hwacha = tile.outer.roccs.collectFirst { case h: Hwacha =>
        topGroup.placeAt(2, context.addHier(h.module))
      }

      context.commit()
    case hwacha: HwachaImp =>
      val context = Floorplan(hwacha)
      val topArray = context.createElasticArray(2)
      val topGroup = context.setTopGroup(topArray)
      // Hwacha floorplan goes here
      context.commit()
  }
}

case object RocketFloorplanAspect
  extends FloorplanAspect[chipyard.TestHarness](
    RocketFloorplans.default
)
```

Listing 5.5: Floorplan generator code for RocketTile modules.

```scala
// See LICENSE for license details
package chipyard.floorplan

import chipyard.TestHarness
import chipyard.{ChipTopLazyRawModuleImp, BuildSystem, DigitalTop}
import freechips.rocketchip.config.{Parameters}
import barstools.floorplan.chisel.{FloorplanAspect, Floorplan}
import barstools.floorplan.chisel.{FloorplanFunction}

object ChipTopFloorplans {

  def default: FloorplanFunction = {
    case top: ChipTopLazyRawModuleImp =>
      val context = Floorplan(top, 1500.0, 1000.0)
      val array = context.createElasticArray(2)
      val topGrid = context.setTopGroup(array)
      val tiles = top.outer.lazySystem match {
        case t: DigitalTop =>
          t.tiles.map(x => context.addHier(x.module))
        case _ =>
          throw new Exception("Unsupported BuildSystem type")
      }
      topGrid.placeAt(1, context.createElasticArray(tiles))
      topGrid.placeAt(0, context.createSpacer(Some("spacer")))
      context.commit()
  }

}

case object ChipTopFloorplanAspect
  extends FloorplanAspect[chipyard.TestHarness](
    ChipTopFloorplans.default orElse
    RocketFloorplans.default
)
```

Listing 5.6: Floorplan generator code for a ChipTop modules.

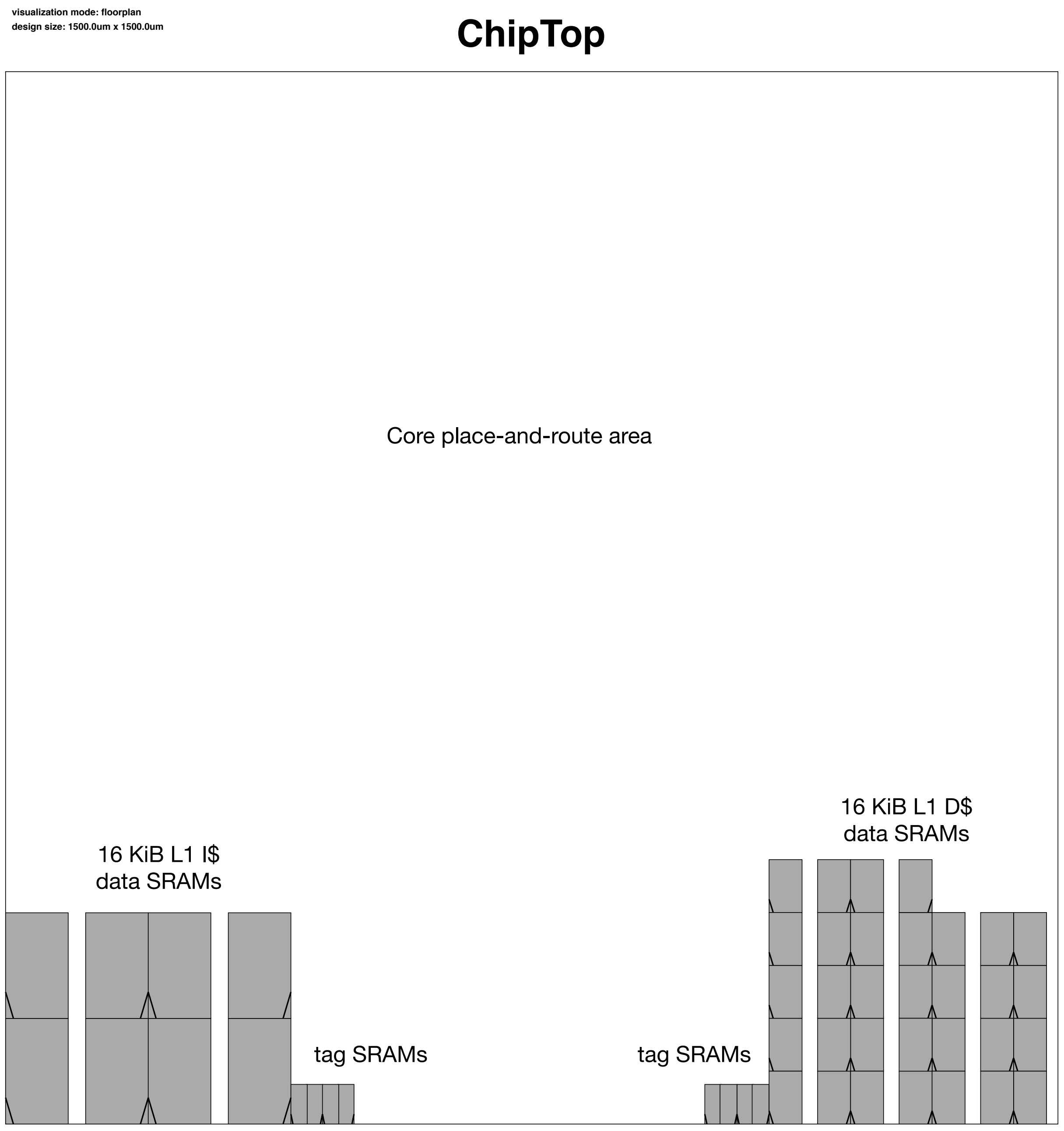

visualization mode: floorplan
design size: 1500.0um x 1500.0um
ChipTop
Core place-and-route area
16 KiB L1 I$
data SRAMs
16 KiB L1 D$
data SRAMs
tag SRAMs
tag SRAMs

Figure 5.8: Annotated floorplan of a single-core ChipTop using the generators shown in Listing 5.6 and Listing 5.5 and visualized using Hammer.

5.6 Future work

The floorplan compiler presented in this dissertation is a promising step towards automating floorplans for generator-based designs. The next step for this project is to be used in the tapeout of a real, complex ASIC design. To achieve this goal, the following work is needed.

- The **HierarchicalBarrier** elements are designed to allow floorplan composability, but they should also be allowed to serve as un-broken hierarchical boundaries for hierarchical place-and-route. With this, the floorplan compiler would be able to separate floorplans for each hierarchical cell as required by most bottom-up hierarchical flows.
- The **GroupAndDedup** pass used to transform the logical hierarchy within FIRRTL is not supported by the floorplan compiler. This pass is not used by all hierarchical designs, but its support would extend the applicability of the floorplan compiler to more complex architectures. To enable this, the floorplan compiler should be able to target **GroupAndDedup** annotations themselves. While FIRRTL does not support annotations of annotations, this can likely be implemented by annotating the module targets with an additional floorplan annotation that contains metadata relating it to the **GroupAndDedup** transformation.
- Modern systems-on-chip are built with many separate power domains which can affect the floorplan. Integration of a framework for adding power domains [109] is needed to improve the utility of the floorplan generation framework.
- The addition of custom element types will allow injection of custom floorplan components that are not implemented by the base framework. This allows the floorplan to be "Future-proof" by adding extensibility, much like Hammer allows arbitrary Tcl code addition via hooks.
- Some Hammer IR primitives, like obstructions, are not currently supported in the floorplan element set. These could be implemented with the custom element types listed above, but most Hammer primitives are fundamental enough to merit direct implementation in the floorplan framework.
- The constraint solving used in the floorplan compiler is rudimentary. Future work should incorporate a more robust constraint solver that will converge over a wider range of inputs.

Chapter 6

Conclusion

This dissertation presents a series of increasingly complex generator-based systems-on-chip designed, manufactured, and tested in 28nm FD-SOI and 16nm FinFET, shown in Figure 6.1, along with a physical design methodology and framework used in their construction. The physical design methodology and framework were developed in conjunction with the systems-on-chip, improving the productivity of small teams of design engineers over the series of chips without sacrificing quality of results (QoR), achieving state-of-the-art general matrix multiply (GEMM) energy efficiency of 209.5 GHFLOPS/W. Based on this success, a floorplanning framework is presented which aims to solve some residual productivity issues stemming from generator-based design flows by enabling generator-based floorplans to be expressed in conjunction with the RTL design.

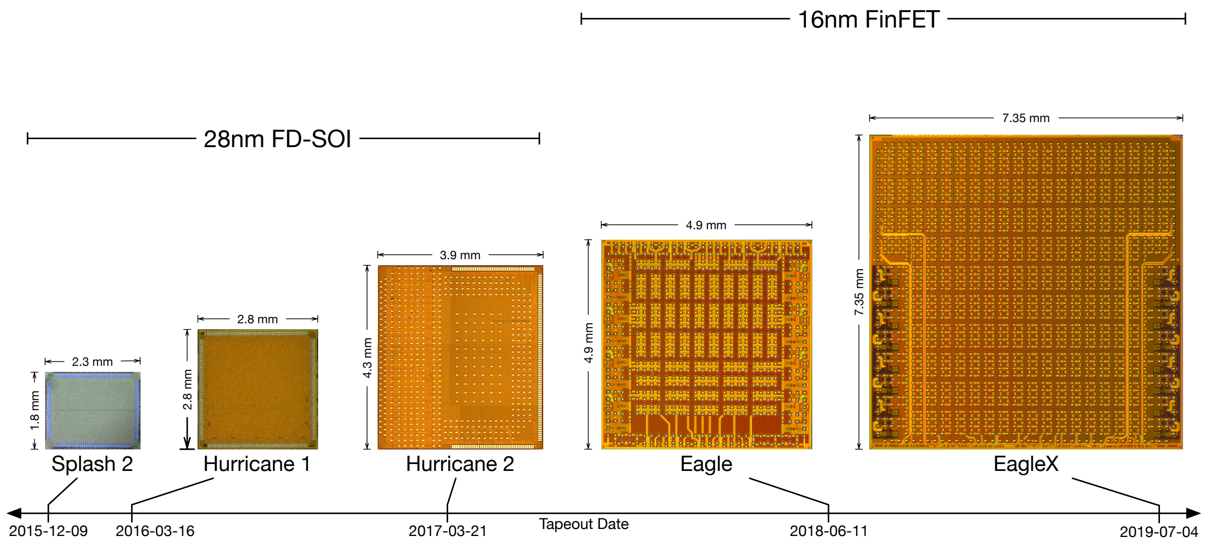


Figure 6.1: Die micrographs of the chips presented in this dissertation to scale.

6.1 Summary of contributions

This work includes the following contributions.

- Serial link design and integration into a digital ASIC spectrometer manufactured in 28nm FD-SOI [51] (Section 2.1).
- Serial link design, integration, and testing of a dual-core RISC-V vector system-on-chip with on-chip fine-grain power management in 28nm FD-SOI [57] (Section 2.2).
- Serial link design, integration, and testing of a single-core RISC-V dual-lane vector system-on-chip with on-chip fine-grain power management in 28nm FD-SOI [81] (Section 2.3).
- An eight-core RISC-V vector machine in 16nm FinFET with state-of-the-art half-precision DGEMM energy efficiency [86] (Section 3.1).
- A 21-core heterogeneous RISC-V vector machine with systolic array accelerator in 16nm FinFET (Section 3.2).
- A physical design framework that enables reusable physical design generators for improving designer productivity [17] (Chapter 4).
- A floorplanning framework in Chisel for reducing the overhead of building floorplans for RTL generator instances (Chapter 5).

6.2 Future work

There remains a large gap between software and hardware design productivity. Further adoption of proven software design principles by hardware designers will close this gap, especially in the areas of physical design and verification. To this end, the following future research is needed.

- This dissertation does not address the verification of generated RTL, which continues to be a challenge to generator-based system design. Of particular significance is verification of generators themselves, which promises greater benefits to productivity than instance-based generation. Further investigation into Agile verification methodology is warranted.
- The EagleX chip is still being tested as of this writing. Early results show significant usability improvements over the Eagle chip, including the ability to run multicore Linux, enabling modern high-level machine learning frameworks to utilize the system's vector acceleration and systolic array. Measurements from EagleX will not only help to validate the system architecture, but will also confirm the efficacy of the generator-based RTL and physical design methodologies presented in this dissertation.

- The Hammer framework is missing important signoff flows like static timing analysis, IR drop analysis, power analysis, and others. Future work should include adding these features, as it will both improve productivity of chip design in general and enable additional work flows that use data generated by these analyses.
- Currently, BAG is not strongly to either Chipyard or Hammer, and most of the BAG-generated circuitry that is included in Chipyard-based designs is integrated manually. There is an opportunity to co-constrain the analog design through BAG with the floorplan implemented with Hammer and the RTL generated by Chisel. Further research is needed to explore the feasibility of this approach and discover possible benefits of this approach.
- Chipyard was recently updated to include the option to implement a ring NoC system bus topology. However, further topology configuration options are needed to improve quality of results on many real systems-on-chip generated using Chipyard. A robust generator framework for networks-on-chip, especially one which includes physical design feedback from generated floorplans, is needed.
- The goal of Chipyard is to include a baseline system-on-chip generator for academic and industrial use. Floorplans for the core set of generators included in Chipyard, like the cores, caches, accelerators, and peripherals, would improve the reusability of these components. Future work should include floorplan generators using the proposed floorplanning framework, ideally with examples implemented in open-source process technologies.
- Machine learning is commonly being applied to electronic design automation problems. Additional research is needed to determine if machine learning can be used effectively in conjunction with the proposed floorplanning and physical design flows.

Bibliography

[1] G. E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 19, 1965), pp. 114–117.

[2] J. G. Spooner. *It’s official: AMD hits 1,000MHz first.* Zdnet. Mar. 2000. URL: `https://www.zdnet.com/article/its-official-amd-hits-1000mhz-first-5000096067/`.

[3] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions”. In: *IEEE Journal of Solid-State Circuits* 9 (5 Oct. 1974), pp. 256–268.

[4] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. “Dark Silicon and the End of Multicore Scaling”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. San Jose, CA, USA: IEEE, June 4–8, 2011, pp. 365–376.

[5] D. Geer. “Chip Makers Turn to Multicore Processors”. In: *Computer* 38.5 (May 2005), pp. 11–13.

[6] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105 (12 Dec. 2017), pp. 2295–2329.

[7] Y.-H. Chen, J. Emer, and V. Sze. “Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators”. In: *IEEE Micro* 37 (3 2017), pp. 12–21.

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. “In-Datacenter Performance Analysis of a

Tensor Processing Unit". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. Toronto, ON, Canada: IEEE, June 24–28, 2017, pp. 1–12.

[9] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolić, I. Stoica, and K. Asanović. *Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures*. Nov. 2019. URL: `http://arxiv.org/pdf/1911.09925v2`.

[10] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtić, S. Bailey, M. Blagojević, P.-F. Chiu, R. Avižienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolić, and K. Asanović. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro* 36 (2 Mar. 2016), pp. 8–20.

[11] S. Bailey. "Rapid ASIC Design for Digital Signal Processors". PhD thesis. EECS Department, University of California, Berkeley, May 2020.

[12] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang. "Creating an Agile Hardware Design Flow". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, July 20–24, 2020, pp. 1–6.

[13] S. Bailey, P. Rigge, J. Han, R. Lin, E. Y. Chang, H. Mao, Z. Wang, C. Markley, A. M. Izraelevitz, A. Wang, N. Narevsky, W. Bae, S. Shauck, S. Montano, J. Norsworthy, M. Razzaque, W. H. Ma, A. Lentiro, M. Doerflein, D. Heckendorn, J. McGrath, F. DeSeta, R. Shoham, M. Stellfox, M. Snowden, J. Cole, D. R. Fuhrman, B. Richards, J. Bachrach, E. Alon, and B. Nikolić. "A Mixed-Signal RISC-V Signal Analysis SoC Generator With a 16-nm FinFET Instance". In: *IEEE Journal of Solid-State Circuits* 54 (10 July 17, 2019), pp. 2786–2801.

[14] C. Celio, P.-F. Chiu, K. Asanović, B. Nikolić, and D. Patterson. "BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS". In: *IEEE Micro* 39 (2 Mar. 2019), pp. 52–60.

[15] S. Galal, O. Shacham, J. S. Brunhaver, J. Pu, A. Vassiliev, and M. Horowitz. "FPU Generator for Design Space Exploration". In: (Apr. 2013).

[16] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40 (4 July 2020), pp. 10–21.

[17] E. Wang, C. Schmidt, A. Izraelevitz, J. Wright, B. Nikolić, E. Alon, and J. Bachrach. "A Methodology for Reusable Physical Design". In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. Santa Clara, CA, USA: IEEE, Mar. 25–26, 2020, pp. 243–249.

[18] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean. "A Graph Placement Methodology for Fast Chip Design". In: *Nature* 594.7862 (June 2021), pp. 207–212.

[19] Z. Or-Bach. *Moore's Law Has Stopped at 28nm*. Mar. 2014. URL: `https://sst.semiconductor-digest.com/2014/03/moores-law-has-stopped-at-28nm/`.

[20] EETimes. *FD SOI Benefits Rise at 14nm*. EETimes. June 13, 2016. URL: `https://www.eetimes.com/fd-soi-benefits-rise-at-14nm/`.

[21] R. Pease and S. Chou. "Lithography and Other Patterning Techniques for Future Electronics". In: *Proceedings of the IEEE* 96.2 (Feb. 2008), pp. 248–270.

[22] Y. Lee and A. Waterman. "Managing Chip Design Complexity in the Domain-Specific SoC Era". In: *2020 IEEE Symposium on VLSI Circuits*. Honolulu, HI, USA: IEEE, June 16–19, 2020, pp. 1–2.

[23] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing". In: *IEEE Micro* 32.5 (Sept. 2012), pp. 38–51.

[24] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing". In: (June 2015).

[25] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. *Manifesto for Agile Software Development*. 2001. URL: `http://www.agilemanifesto.org/`.

[26] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014.

[27] K. Asanović, R. Avižienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016.

[28] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing Hardware in a Scala Embedded language". In: *DAC Design Automation Conference 2012*. San Francisco, CA, USA: IEEE, July 3–7, 2012, pp. 1212–1221.

[29] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. “Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Irvine, CA, USA: IEEE, Nov. 13–16, 2017, pp. 209–216.

[30] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolić, R. Katz, J. Bachrach, and K. Asanović. “Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. Los Angeles, California: IEEE Press, 2018, pp. 29–42.

[31] A. B. Kahng. “Machine Learning Applications in Physical Design: Recent Results and Directions”. In: *Proceedings of the 2018 International Symposium on Physical Design*. ISPD ’18. Monterey, California, USA: Association for Computing Machinery, 2018, pp. 68–73.

[32] E. Chang, J. Han, W. Bae, Z. Wang, N. Narevsky, B. Nikolić, and E. Alon. “BAG2: A Process-Portable Framework for Generator-Based AMS Circuit Design”. In: *2018 IEEE Custom Integrated Circuits Conference (CICC)*. San Diego, CA, USA: IEEE, Apr. 8–11, 2018, pp. 1–8.

[33] A. Gonzalez, J. Zhao, B. Korpan, H. Genc, C. Schmidt, J. Wright, A. Biswas, A. Amid, F. Sheikh, A. Sorokin, S. Kale, M. Yalamanchi, R. Yarlagadda, M. Flannigan, L. Abramowitz, E. Alon, Y. S. Shao, K. Asanović, and B. Nikolić. “A 16mm^2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET”. In: *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)*. 2021.

[34] K. Settaluri, A. Haj-Ali, Q. Huang, K. Hakhamaneshi, and B. Nikolić. “AutoCkt: Deep Reinforcement Learning of Analog Circuit Designs”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (Mar. 2020).

[35] *IEEE Standard for Universal Verification Methodology Language Reference Manual.* 1800.2-2017. IEEE.

[36] E. Seligman, T. Schubert, and M. V. A. K. Kumar. *Formal Verification*. Elsevier Science & Techn., July 24, 2015. 408 pp.

[37] L. Truong, S. Herbst, R. Setaluri, M. Mann, R. Daly, K. Zhang, C. Donovick, D. Stanley, M. Horowitz, C. Barrett, and P. Hanrahan. “fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components”. In: *Computer Aided Verification*. Springer International Publishing, 2020, pp. 403–414.

[38] B. West. *Hierarchy Management for Million Plus Gate Counts*. Design & Reuse. URL: `https://www.design-reuse.com/articles/5171/hierarchy-management-for-million-plus-gate-counts.html`.

[39] I. L. Markov, J. Hu, and M.-C. Kim. “Progress and Challenges in VLSI Placement Research”. In: *Proceedings of the IEEE* 103.11 (Nov. 2015), pp. 1985–2003.

[40] B. Bredthauer, M. Olbrich, and E. Barke. “STP - A Quadratic VLSI Placement Tool Using Graphic Processing Units”. In: *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)* (June 25–28, 2018). Geneva, Switzerland: IEEE, June 25–28, 2018, pp. 77–84.

[41] A. Al-Kawam and H. M. Harmanani. “A Parallel GPU Implementation of the Timber Wolf Placement Algorithm”. In: *2015 12th International Conference on Information Technology - New Generations* (Apr. 13–15, 2015). Las Vegas, NV, USA: IEEE, Apr. 13–15, 2015, pp. 792–795.

[42] Y. Zhou, Y. Yan, and W. Yan. “A Method to Speed up VLSI Hierarchical Physical Design in Floorplanning”. In: *2017 IEEE 12th International Conference on ASIC (ASICON)* (Oct. 25–28, 2017). Guiyang, China: IEEE, Oct. 25–28, 2017, pp. 347–350.

[43] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang. “RePlAce: Advancing Solution Quality and Routability Validation in Global Placement”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38 (9 Sept. 2019), pp. 1717–1730.

[44] B. Li and P. D. Franzon. “Machine Learning in Physical Design”. In: *2016 IEEE 25th Conference on Electrical Performance Of Electronic Packaging And Systems (EPEPS)* (Oct. 23–26, 2016). San Diego, CA, USA: IEEE, Oct. 23–26, 2016, pp. 147–150.

[45] V. A. Chhabria, Y. Zhang, H. Ren, B. Keller, B. Khailany, and S. S. Sapatnekar. *MAVIREC: ML-Aided Vectored IR-DropEstimation and Classification*. Dec. 2020. URL: `http://arxiv.org/pdf/2012.10597v1`.

[46] H. Ren, G. F. Kokai, W. J. Turner, and T.-S. Ku. “ParaGraph: Layout Parasitics and Device Parameter Prediction using Graph Neural Networks”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)* (July 20–24, 2020). San Francisco, CA, USA: IEEE, July 20–24, 2020, pp. 1–6.

[47] L. Bai and L. Chen. “Machine-Learning-Based Early-Stage Timing Prediction in SoC Physical Design”. In: *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)* (Oct. 31–Nov. 3, 2018). Qingdao, China: IEEE, Oct. 31–Nov. 3, 2018, pp. 1–3.

[48] Y.-C. Lu, J. Lee, A. Agnesina, K. Samadi, and S. K. Lim. "GAN-CTS: A Generative Adversarial Framework for Clock Tree Prediction and Optimization". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (Nov. 4–7, 2019). Westminster, CO, USA: IEEE, Nov. 4–7, 2019, pp. 1–8.

[49] J. Cooley. *Nvidia uses MENT Nitro Physical Floorplanning for ICC/ICC2*. DeepChip. Apr. 4, 2017. URL: `http://www.deepchip.com/items/0570-02.html`.

[50] D. Jacquet, F. Hasbani, P. Flatresse, R. Wilson, F. Arnaud, G. Cesana, T. D. Gilio, C. Lecocq, T. Roy, A. Chhabra, C. Grover, O. Minez, J. Uginet, G. Durieu, C. Adobati, D. Casalotto, F. Nyer, P. Menut, A. Cathelin, I. Vongsavady, and P. Magarshack. "A 3 GHz Dual Core Processor ARM Cortex TM -A9 in 28 nm UTBB FD-SOI CMOS With Ultra-Wide Voltage Range and Energy Efficiency Optimization". In: *IEEE Journal of Solid-State Circuits* 49.4 (Apr. 2014), pp. 812–826.

[51] S. Bailey, J. Wright, N. Mehta, R. Hochman, R. Jarnot, V. Milovanović, D. Werthimer, and B. Nikolić. "A 28nm FDSOI 8192-Point Digital ASIC Spectrometer from a Chisel Generator". In: *2018 IEEE Custom Integrated Circuits Conference (CICC)*. San Diego, CA, USA: IEEE, Apr. 8–11, 2018, pp. 1–4.

[52] L. Dong, M. Wang, and S. Shi. "A New Digital Spectrometer for Low Frequency Solar Radio Observation Based on FPGA". In: *2010 2nd International Conference on Signal Processing Systems*. IEEE, July 2010.

[53] J. Wright. "Design of a Lightweight Serial Link Generator for Test Chips". Master's thesis. EECS Department, University of California, Berkeley, Dec. 2017.

[54] N. Mehta, C. Sun, M. Wade, and V. Stojanović. "A Differential Optical Receiver With Monolithic Split-Microring Photodetector". In: *IEEE Journal of Solid-State Circuits* 54 (8 Aug. 2019), pp. 2230–2242.

[55] F. Hsiao, A. Tang, Y. Kim, B. Drouin, G. Chattopadhyay, and M.-C. F. Chang. "A 2.2 GS/s 188mW Spectrometer Processor in 65nm CMOS for Supporting Low-Power THz Planetary Instruments". In: *2015 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, Sept. 2015.

[56] B. Richards, N. Nicolici, H. Chen, K. Chao, R. Abiad, D. Werthimer, and B. Nikolić. "A 1.5GS/s 4096-Point Digital Spectrum Analyzer for Space-Borne Applications". In: *2009 IEEE Custom Integrated Circuits Conference*. IEEE, Sept. 2009.

[57] J. Wright, C. Schmidt, B. Keller, D. P. Dabbelt, J. Kwak, V. Iyer, N. Mehta, P.-F. Chiu, S. Bailey, K. Asanović, and B. Nikolić. "A Dual-Core RISC-V Vector Processor With On-Chip Fine-Grain Power Management in 28-nm FD-SOI". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28 (12 Dec. 2020), pp. 2721–2725.

[58] A. Villas-Boas. *Two Simple but Major Problems with Smartphones Need to be Figured out Before Companies Focus on Making Foldable Smartphones.* Business Insider. Mar. 11, 2020. URL: `https://www.businessinsider.com/smartphone-battery-life-durability-foldable-devices-samsung-galaxy-iphone-2020-3`.

[59] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. "A Dynamic Voltage Scaled Microprocessor System". In: *IEEE Journal of Solid-State Circuits* 35 (11 Nov. 2000), pp. 1571–1580.

[60] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevtić, B. Keller, S. Bailey, M. Blagojević, P.-F. Chiu, H.-P. Le, P.-H. Chen, N. Sutardja, R. Avižienis, A. Waterman, B. Richards, P. Flatresse, E. Alon, K. Asanović, and B. Nikolić. "A RISC-V Vector Processor with Tightly-Integrated Switched-Capacitor DC-DC Converters in 28nm FDSOI". In: *2015 Symposium on VLSI Circuits (VLSI Circuits)* (June 17–19, 2015). Kyoto, Japan: IEEE, June 17–19, 2015, pp. C316–C317.

[61] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony. "2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products". In: *2020 IEEE International Solid- State Circuits Conference - (ISSCC).* San Francisco, CA, USA: IEEE, Feb. 16–20, 2020, pp. 44–45.

[62] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge". In: *IEEE Micro* 32 (2 Mar. 2012), pp. 20–27.

[63] B. Zimmer, Y. Lee, A. Puggelli, J. Kwak, R. Jevtić, B. Keller, S. Bailey, M. Blagojević, P.-F. Chiu, H.-P. Le, P.-H. Chen, N. Sutardja, R. Avižienis, A. Waterman, B. Richards, P. Flatresse, E. Alon, K. Asanović, and B. Nikolić. "A RISC-V Vector Processor With Simultaneous-Switching Switched-Capacitor DC–DC Converters in 28 nm FDSOI". In: *IEEE Journal of Solid-State Circuits* 51 (4 Apr. 2016), pp. 930–942.

[64] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* San Diego, CA, USA: IEEE, Dec. 5, 2003, pp. 7–18.

[65] B. Keller, M. Cochet, B. Zimmer, Y. Lee, M. Blagojević, J. Kwak, A. Puggelli, S. Bailey, P.-F. Chiu, P. Dabbelt, C. Schmidt, E. Alon, K. Asanović, and B. Nikolić. "Sub-Microsecond Adaptive Voltage Scaling in a 28nm FD-SOI Processor SoC". In: *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference* (Sept. 12–15, 2016). Lausanne, Switzerland: IEEE, Sept. 12–15, 2016, pp. 269–272.

[66] J. Kwak and B. Nikolić. "A 550–2260MHz Self-Adjustable Clock Generator in 28nm FDSOI". In: *2015 IEEE Asian Solid-State Circuits Conference (A-SSCC).* Xiamen, China: IEEE, Nov. 9–11, 2015, pp. 1–4.

[67] M. Cochet, B. Keller, S. Clerc, F. Abouzeid, A. Cathelin, J.-L. Autran, P. Roche, and B. Nikolić. “A 225 µm ^{2} Probe Single-Point Calibration Digital Temperature Sensor Using Body-Bias Adjustment in 28 nm FD-SOI CMOS”. In: *IEEE Solid-State Circuits Letters* 1 (1 Jan. 2018), pp. 14–17.

[68] M. Blagojević, M. Cochet, B. Keller, P. Flatresse, A. Vladimirescu, and B. Nikolić. “A Fast, Flexible, Positive and Negative Adaptive Body-Bias Generator in 28nm FDSOI”. In: *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. Honolulu, HI, USA: IEEE, June 15–17, 2016, pp. 1–2.

[69] C. Schmidt and A. Ou. “Hwacha: A Data-Parallel RISC-V Extension and Implementation”. RISC-V Summit. Dec. 2018.

[70] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanović. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1.* Tech. rep. UCB/EECS-2015-262. EECS Department, University of California, Berkeley, Dec. 2015.

[71] R. M. Russell. “The CRAY-1 Computer System”. In: *Communications of the ACM* 21.1 (Jan. 1978), pp. 63–72.

[72] C. Lomont. *Introduction to Intel Advanced Vector Extensions. Intel White Paper.* 2011. URL: `https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html`.

[73] Y. Lee, A. Ou, C. Schmidt, S. Karandikar, H. Mao, and K. Asanović. *The Hwacha Microarchitecture Manual, Version 3.8.1.* Tech. rep. UCB/EECS-2015-263. EECS Department, University of California, Berkeley, Dec. 2015.

[74] D. C. Snowdon, S. M. Petters, and G. Heiser. “Accurate On-Line Prediction of Processor and Memoryenergy Usage under Voltage Scaling”. In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software.* EMSOFT ’07. Salzburg, Austria: Association for Computing Machinery, 2007, pp. 84–93.

[75] S. Eyerman and L. Eeckhout. “Fine-Grained DVFS Using on-Chip Regulators”. In: *ACM Trans. Archit. Code Optim.* 8.1 (Feb. 2011).

[76] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu. “A Study on the Use of Performance Counters to Estimate Power in Microprocessors”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 60 (12 Dec. 2013), pp. 882–886.

[77] B. Keller, M. Cochet, B. Zimmer, J. Kwak, A. Puggelli, Y. Lee, M. Blagojević, S. Bailey, P.-F. Chiu, P. Dabbelt, C. Schmidt, E. Alon, K. Asanović, and B. Nikolić. “A RISC-V Processor SoC With Integrated Power Management at Submicrosecond Timescales in 28 nm FD-SOI”. In: *IEEE Journal of Solid-State Circuits* 52 (7 July 2017), pp. 1863–1875.

[78] T. Webel, P. M. Lobo, R. Bertran, G. M. Salem, M. Allen-Ware, R. Rizzolo, S. M. Carey, T. Strach, A. Buyuktosunoglu, C. Lefurgy, P. Bose, R. Nigaglioni, T. Slegel, M. S. Floyd, and B. W. Curran. “Robust Power Management in the IBM z13”. In: *IBM Journal of Research and Development* 59 (4/5 July 2015), 16:1–16:12.

[79] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. "Power Management Architecture of the 2nd Generation Intel Core Microarchitecture, Formerly Codenamed Sandy Bridge". In: *2011 IEEE Hot Chips 23 Symposium (HCS)*. Stanford, CA, USA: IEEE, Aug. 17–19, 2011, pp. 1–33.

[80] B. Keller. "Energy-Efficient System Design Through Adaptive Voltage Scaling". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2019.

[81] C. Schmidt, A. Amid, J. Wright, B. Keller, H. Mao, K. Settaluri, J. Salomaa, J. Zhao, A. Ou, K. Asanović, and B. Nikolić. "Programmable Fine-Grained Power Management and System Analysis of RISC-V Vector Processors in 28-nm FD-SOI". In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 210–213.

[82] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill. "FIVR — Fully Integrated Voltage Regulators on 4th Generation Intel Core SoCs". In: *2014 IEEE Applied Power Electronics Conference and Exposition - APEC 2014*. Fort Worth, TX, USA: IEEE, Mar. 16–20, 2014, pp. 432–439.

[83] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst. "14.5 Envision: A 0.26-to-10TOPS/W Subword-Parallel Dynamic-Voltage-Accuracy-Frequency-Scalable Convolutional Neural Network Processor in 28nm FDSOI". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. San Francisco, CA, USA: IEEE, Feb. 5–7, 2017, pp. 246–247.

[84] D. Hisamoto, W.-C. Lee, J. Kedzierski, H. Takeuchi, K. Asano, C. Kuo, E. Anderson, T.-J. King, J. Bokor, and C. Hu. "FinFET-a Self-Aligned Double-Gate MOSFET Scalable to 20 nm". In: *IEEE Transactions on Electron Devices* 47.12 (2000), pp. 2320–2325.

[85] I. Cutress. *Where are my GAA-FETs? TSMC to stay with FinFET for 3nm*. AnandTech. Aug. 26, 2020. URL: `https://www.anandtech.com/show/16041/where-are-my-gaafets-tsmc-to-stay-with-finfet-for-3nm`.

[86] C. Schmidt, J. Wright, Z. Wang, E. Chang, A. Ou, W. Bae, S. Huang, A. Flynn, B. Richards, K. Asanović, E. Alon, and B. Nikolić. "4.3 An Eight-Core 1.44GHz RISC-V Vector Machine in 16nm FinFET". In: *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 64. San Francisco, CA, USA: IEEE, Feb. 13–22, 2021, pp. 58–60.

[87] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo. "7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16". In: *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. San Francisco, CA, USA: IEEE, Feb. 17–21, 2019, pp. 142–144.

[88] P. N. Whatmough, S. K. Lee, M. Donato, H.-C. Hsueh, S. Xi, U. Gupta, L. Pentecost, G. G. Ko, D. Brooks, and G.-Y. Wei. "A 16nm 25mm^2 SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators". In: *2019 Symposium on VLSI Circuits*. Kyoto, Japan: IEEE, June 9–14, 2019, pp. C34–C35.

[89] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. "A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS". In: *2019 Symposium on VLSI Circuits*. Kyoto, Japan: IEEE, July 9, 2019–July 14, 2017, pp. C30–C31.

[90] A. Rylyakov, J. Tierno, G. English, D. Friedman, and M. Meghelli. "A Wide Power-Supply Range (0.5V-to-1.3V) Wide Tuning Range (500 MHz-to-8 GHz) All-Static CMOS AD PLL in 65nm SOI". In: *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, Feb. 2007.

[91] K. Lee, S.-J. Lee, and H.-J. Yoo. "Low-Power Network-on-Chip for High-Performance SoC Design". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14 (2 Feb. 2006), pp. 148–160.

[92] S. Murali, L. Benini, and G. D. Micheli. "An Application-Specific Design Methodology for On-Chip Crossbar Generation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26 (7 July 2007), pp. 1283–1296.

[93] R. Kirby, S. Godil, R. Roy, and B. Catanzaro. "CongestionNet: Routing Congestion Prediction Using Deep Graph Neural Networks". In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)* (Oct. 6–9, 2019). Cuzco, Peru: IEEE, Oct. 6–9, 2019, pp. 217–222.

[94] H. Kung. "Why Systolic Architectures?" In: *Computer* 15.1 (Jan. 1982), pp. 37–46.

[95] R. Smith and J. Ho. *The Apple iPhone 6s and iPhone 6s Plus Review*. AnandTech. Nov. 2015. URL: `https://www.anandtech.com/show/9686/the-apple-iphone-6s-and-iphone-6s-plus-review/3`.

[96] W. Dally and A. Chang. "The role of custom design in ASIC chips". In: *Proceedings 37th Design Automation Conference* (June 5–9, 2000). Los Angeles, CA, USA: IEEE, June 5–9, 2000, pp. 643–647.

[97] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and ¡0.5MB Model Size". In: (Feb. 2016).

[98] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer. "SqueezeNext: Hardware-Aware Neural Network Design". In: *Design Automation Conference 2018 (and CVPR 2018 workshop)* (Mar. 2018).

[99] B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer. "INVITED: A Modular Digital VLSI Flow for High-Productivity SoC Design". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)* (June 24–28, 2018). San Francisco, CA, USA: IEEE, June 24–28, 2018, pp. 1–6.

[100] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça, et al. "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain". In: *Proc. GOMACTECH* (2019), pp. 1105–1110.

[101] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric. "ASAP7: A 7-nm FinFET Predictive Process Design Kit". In: *Microelectronics Journal* 53 (2016), pp. 105–115.

[102] S. Burns. *FIRRTL Pass for Area and Timing*. Chisel Community Conference. Nov. 2018. URL: `https://www.youtube.com/watch?v=FktjrjRVBoY`.

[103] S. Hashemi, C.-T. Ho, A. B. Kahng, H.-Y. Liu, and S. Reda. "METRICS 2.0: A Machine-Learning BasedOptimization System for IC Design". In: *Workshop on Open-Source EDA Technology 2018 (WOSET-2018)*. 21. Nov. 2018.

[104] M. A. Kabir and Y. Peng. "Holistic Chiplet–Package Co-Optimization for Agile Custom 2.5-D Design". In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* 11 (5 May 2021), pp. 715–726.

[105] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[106] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In: *ECOOP'97 — Object-Oriented Programming*. Springer Berlin Heidelberg, 1997, pp. 220–242.

[107] A. Izraelevitz. "Unlocking Design Reuse with Hardware Compiler Frameworks". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2019.

[108] H. Cook, W. Terpstra, and Y. Lee. "Diplomatic Design Patterns: A TileLink Case Study". In: *Proceedings of First Workshop on Computer Architecture Research with RISC-V*. CARRV '17. Boston, MA, Oct. 2017.

[109] A. Nayak, K. Zhang, R. Setaluri, A. Carsello, M. Mann, S. Richardson, R. Bahr, P. Hanrahan, M. Horowitz, and P. Raina. "A Framework for Adding Low-Overhead, Fine-Grained Power Domains to CGRAs". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2020.