An Agile Approach to Building RISC-V Microprocessors

Yunsup Lee
Andrew Waterman
Henry Cook
Brian Zimmer
Ben Keller
Alberto Puggelli
Jaehwa Kwak
Ružica Jevtić
Stevo Bailey
Milovan Blagojević
Pi-Feng Chiu
Rimas Avižienis
Brian Richards
Jonathan Bachrach
David Patterson
Elad Alon
Borivoje Nikolić
Krste Asanović

The authors adopted an agile hardware development methodology for 11 RISC-V microprocessor tape-outs on 28-nm and 45-nm CMOS processes. This enabled small teams to quickly build energy-efficient, cost-effective, and competitive high-performance microprocessors. The authors present a case study of one prototype featuring a RISC-V vector microprocessor integrated with switched-capacitor DC–DC converters and an adaptive clock generator in a 28-nm, fully depleted silicon-on-insulator process.

The era of rapid transistor performance improvement is coming to an end, increasing pressure on architects and circuit designers to improve energy efficiency. Modern systems on chip (SoCs) incorporate a large and growing number of specialized hardware units to execute specific tasks efficiently, often organized into multiple dynamic voltage and clock frequency domains to further save energy under varying computational loads. This growing complexity has led to a design productivity crisis, stimulating development of new tools and methodologies to enable the completion of complex chip designs on schedule and within budget.

We propose leveraging lessons learned from the software world by applying aspects of the software agile development model to hardware. Traditionally, software was developed via the waterfall development model, a sequential process that consists of distinct phases that rigidly follow one another, just as is done for hardware design today. Overbudget, late, and abandoned software projects were commonplace, motivating a revolution in software development, demarcated by the publication of *The Agile Manifesto* in 2001.[1] The philosophy of agile software development emphasizes individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. In practice, the agile approach leads to small teams iteratively refining a set of working-but-incomplete prototypes until the end result is acceptable.

Inspired by the positive disruption of *The Agile Manifesto* on software development, we propose a set of principles to guide a new agile hardware development methodology. Our proposal is informed by our experiences as a small group of researchers designing and fabricating 11 processor chips in five years. Lacking the massive resources of industrial design teams, we were forced to abandon standard industry practices and explore different approaches to hardware design. We detail this approach's benefits with a case study of one of these chips, Raven-3, which achieved

0272-1732/16/$33.00 © 2016 IEEE

groundbreaking energy efficiency by integrating a novel RISC-V vector architecture with efficient on-chip DC–DC converters. Taken together, our experiences developing these academic prototype chips make a powerful argument for the promise of agile hardware design in the post-Moore's law era.

An Agile Hardware Manifesto

Borrowing heavily from the agile software development manifesto, we propose the following principles for hardware design:

- *Incomplete, fabricatable prototypes* over fully featured models.
- *Collaborative, flexible teams* over rigid silos.
- *Improvement of tools and generators* over improvement of the instance.
- *Response to change* over following a plan.

Here, we explain the importance of these principles for building hardware and contrast them with the original principles of agile software development.

Incomplete, Fabricatable Prototypes over Fully Featured Models

We believe the primary benefit of adopting the agile model for hardware design is that it lets us drastically reduce the cost of verification and validation by iterating through many working prototypes of our designs. In this article, we use the standard definition of *verification* as testing that determines whether each component and the assembly of components correctly meets its specification ("Are we building the thing right?"). In hardware design, the term *validation* is usually narrowly applied to postsilicon testing to ensure that manufactured parts operate within design parameters. We use the broader and, to our minds, more useful definition of validation from the software world, in which validation ensures that the product serves its intended purposes ("Are we building the right thing?").

Figure 1 contrasts the agile hardware development model with the waterfall model. The waterfall model, shown in Figure 1a, relies on Gantt charts, high-level architecture models, rigid processes such as register-transfer level (RTL) freezes, and CPU centuries of simulations in an attempt to achieve a single viable design point. In our agile hardware methodology, we first push a trivial prototype with a minimal working feature set all the way through the toolflow to a point where it could be taped out for fabrication. We refer to this tape-out-ready design as a *tape-in*. Then we begin adding features iteratively, moving from one fabricatable tape-in to the next as we add features. The agile hardware methodology will always have an available tape-out candidate with some subset of features for any tape-out deadline, whereas the conventional waterfall approach is in danger of grossly overshooting any tape-out deadline because of issues at any stage of the process.

Although the agile methodology provides some benefits in terms of verification effort, it particularly shines at validation; agile designs are more likely to meet energy and performance expectations using the iterative process. Unlike conventional approaches that use high-level architectural models of the whole design for validation with the intent to later refine these models into an implementation, we emphasize validation using a full RTL implementation of each incomplete prototype with the intent to add features if there is time. Unlike high-level architectural models, the actual RTL is guaranteed to be cycle accurate. Furthermore, the actual RTL design can be pushed through the entire very large-scale integration (VLSI) toolflow to give early feedback on back-end physical design issues that will affect the quality of results (QoR), including cycle time, area, and power. For example, Figure 1b shows a feature F1 being reworked into feature F1′ after validation showed that it would not meet performance or QoR goals, whereas feature F3 was dropped after validation showed that it exceeded physical design budgets or did not add sufficient value to the end system. These decisions on F1 and F3 can be informed by the results of the attempted physical design before completing the final feature F4.

Figure 1c illustrates our iterative approach to validation of a single prototype. Each circle's circumference represents the relative time and effort it takes to begin validating a design using a particular methodology. Although the latency and cost of these

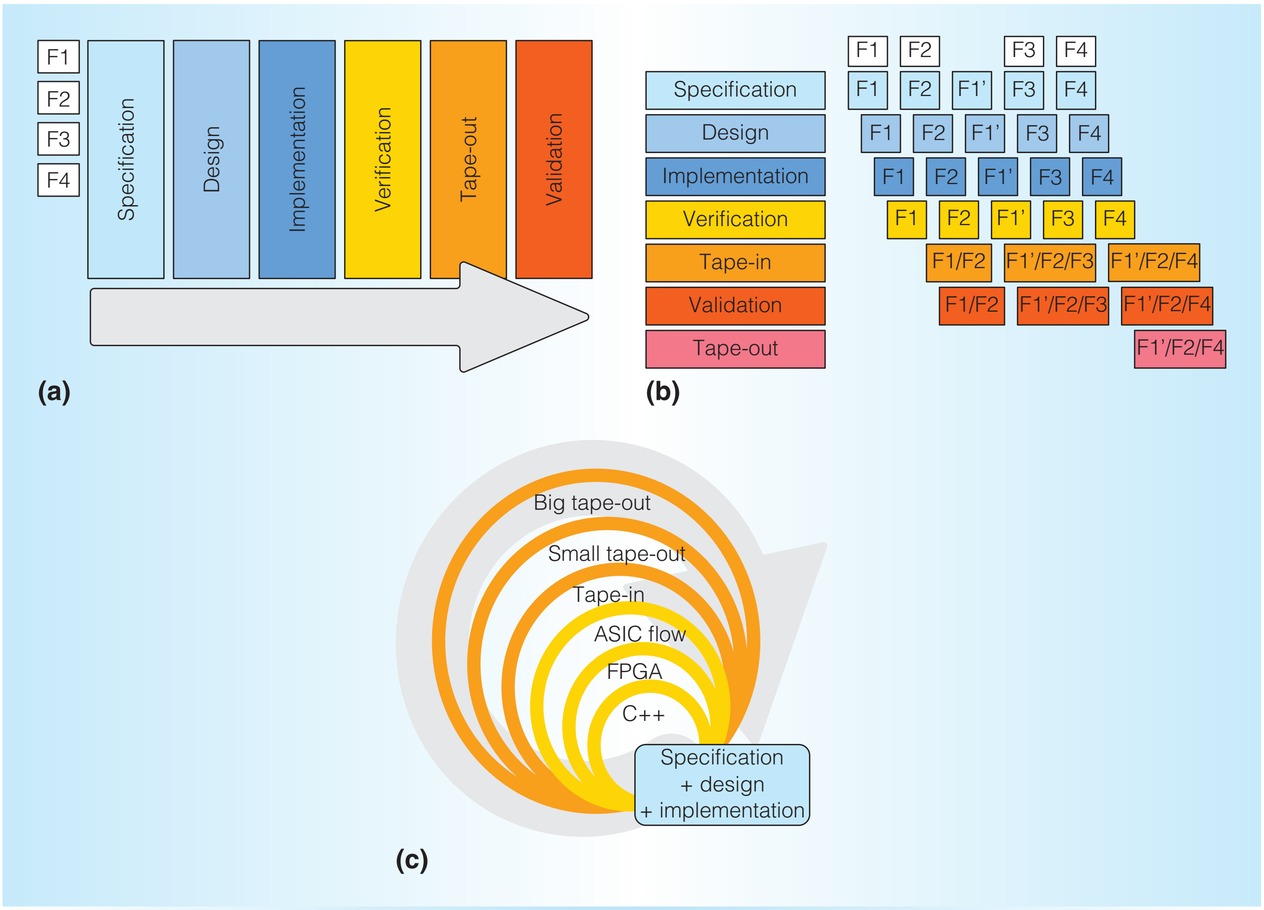


Figure 1. Contrasting the agile and waterfall models of hardware design. The labels F*N* represent various desired features of the design. (a) The waterfall model steps all features through each activity sequentially, producing a tape-out candidate only when all features are complete. (b) The agile model adds features incrementally, resulting in incremental tape-out candidates as individual features are completed, reworked, or abandoned. (c) As validation progresses, designs are subjected to lengthier and more accurate evaluation methodologies.

prototype-modeling efforts increase toward the outer circles, our confidence in the validation results produced increases in tandem. Using fabricatable prototypes increases validation bandwidth, because a complete RTL design can be mapped to field-programmable gate arrays (FPGAs) to run end-application software stacks orders of magnitude faster than with software simulators. In agile hardware development, the FPGA models of iterative prototype RTL designs (together with accompanying QoR numbers from the VLSI toolflow) fulfill the same function as working prototypes in agile software development, providing a way for customers to give early and frequent feedback to validate design choices. This enables customer collaboration as envisioned in the original agile software manifesto.

Collaborative, Flexible Teams over Rigid Silos

Traditional hardware design teams usually are organized around the waterfall model, with engineers assigned to particular functions, such as architecture specification, microarchitecture design, RTL design, verification, physical design, and validation. This specialization of skills is more extreme than for pure software development, as reflected in the specificity of designers' job titles (architect, hardware engineer, or verification engineer). Sometimes, entire functions, such as back-end physical design, are even outsourced to different companies. As the design progresses through these functional stages, extensive effort is required to communicate design intent for each block, and consequently to understand the documentation received from the preceding stage. Misunderstandings can lead to subtle errors that require extensive verification to prevent, and QoR suffers because no engineer views the whole flow. End-to-end design iterations are too expensive to contemplate. Considerable effort is required to push high-level changes down through the implementation hierarchy, particularly across company boundaries. This leads to innovation-stunting practices such as the RTL freeze, in which only show-stopping bugs are allowed to unfreeze a design. In addition, balancing the workload across the different stages is difficult because engineers focused in one functional area often lack the training and experience to be effective in other roles.

In the agile hardware model, engineers work in collaborative, flexible teams that can drive a feature through all implementation stages, avoiding most of the documentation overhead required to map a feature to the hardware. Each team can iterate on the high-level architectural design with full visibility into low-level physical implementation and can validate system-level software impacts on an intermediate prototype. Figure 1b shows how multiple teams can work on different features in a pipelined fashion, each iterating through multiple levels of abstraction (as shown in Figure 1c). A given tape-in will accumulate some small number of feature updates and integrate these for whole-system validation. Engineers naturally develop a complete vertical skill set, and the focus of project management is on prioritizing features for implementation, not communication between silos.

Improvement of Tools and Generators over Improvement of the Instance

Modern software systems could not be built economically without the extreme reuse enabled by extensive software libraries written in modern programming languages, or without functioning compilers and an automated build process. In contrast, most commercial hardware design flows rely on rather primitive languages to describe hardware, and frequent manual intervention on tool outputs to work around tool bugs and long tool runtimes. The poor languages and buggy tools hinder development of truly reusable hardware components and lead to a focus on building a single instance instead of designing components that can be easily reused in future designs.

An agile hardware methodology relies on better hardware description languages (HDLs) to enable reuse. Instead of constructing an optimized instance, engineers develop highly parameterized generators that can be used to automatically explore a design space or to support future use in a different design. Better reuse is also the primary way in which an agile hardware methodology can reduce verification costs, because the verification infrastructure of these generators is also portable across designs.

An agile hardware methodology also strives to eliminate manual intervention in the chip's build process. Replacing or fixing the many commercial CAD tools required to complete a chip design is not practical, but most of a chip's flow can be effectively automated with sufficient effort. Perhaps the most labor-intensive part of a modern chip flow is dealing with physical design issues for a new technology node. However, most new chip designs are in older technologies, because only a few products can justify the cost of using an advanced node. Also, as technology scaling continues to slow, back-end flows and CAD tools will have time to mature and should become more automatable, even for the most advanced nodes. The end of scaling will also coincide with a greater need for a variety of specialized designs, where the agile model should show greater benefit.

Our emphasis on tools and generators is not at odds with the agile software development manifesto, which emphasizes collaboration over tools. Our proposed methodology also values collaboration more than specific tools, but we feel that modern hardware design is far less automated and exhibits far less reuse than modern software development (or even software development in 2001). Accordingly, there is a need to emphasize the importance of reuse and automation enabled by new tools in the hardware design sphere, because few modern hardware design teams emphasize these universally accepted software principles.

Response to Change over Following a Plan

This principle mirrors the original manifesto. Unlike software, a chip design will ultimately be frozen into a mask set and mass-manufactured, with no scope for frequent updates. However, even over the timeframe of a single chip's development, requirements will change owing to market shifts, customer feature additions, or standards-body deliberations. In agile design, change can result from validation testing (as opposed to the waterfall model, in which designs will only change in response to verification problems).

An agile hardware development process embraces continual change in chip specifications as a means to enhance competitiveness. Every incremental fabricatable prototype, not only the most recent, becomes a starting point for the changed set of design requirements, with any change treated as a reprioritization of features to implement, drop, or rework. By first implementing features that are unlikely to change, designers minimize the effort lost to late changes.

Implementing the Agile Hardware Methodology

Here, we describe the steps we took toward implementing an agile hardware design process. All of our processors, both general purpose and specialized, were built on the free and open RISC-V instruction architecture (ISA). We developed our RTL source code using Chisel (Constructing Hardware in a Scala-Embedded Language), an open-source hardware construction language, and expressed the code as libraries of highly parameterized hardware generators. We developed a highly automated back-end design flow to reduce manual effort in pushing designs through to layout. Finally, we present the resulting chips from our agile hardware design process.

RISC-V

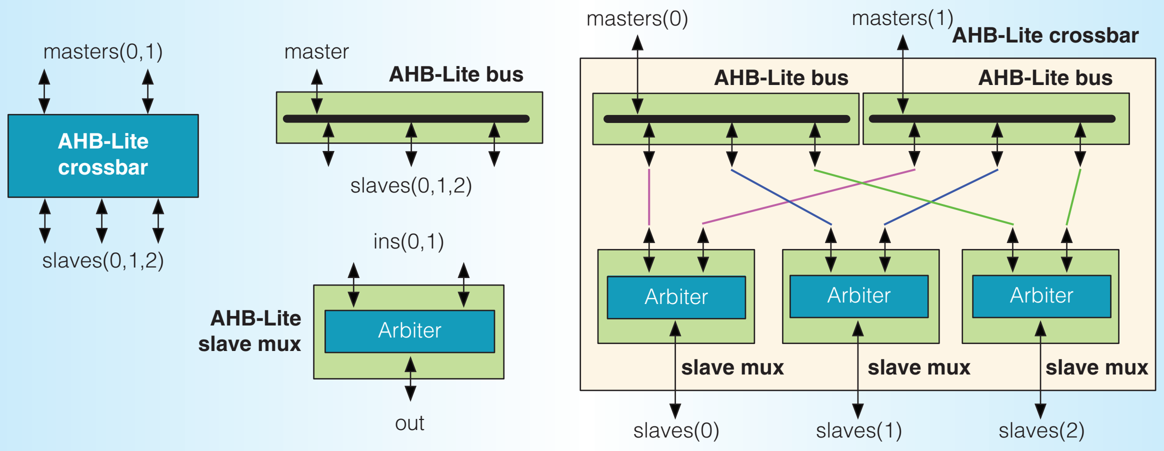
RISC-V is a free, open, and extensible ISA that forms the basis of all of our programmable processor designs.[2] Unlike many earlier efforts that designed open processor cores, RISC-V is an ISA specification, intended to allow many different hardware implementations to leverage common software development. RISC-V is a modular architecture, with variants covering 32-, 64-, and 128-bit address spaces. The base integer instruction set is lean, requiring fewer than 50 user-level hardware instructions to support a full modern software stack, which enables microprocessor designers to quickly bring up fully functional prototypes and add additional features incrementally. In addition to simplifying the implementation of new microarchitectures, the RISC-V design provides an ideal base for custom accelerators. Not only can accelerators reuse common control-processor implementations, they can share a single software stack, including the compiler toolchain and operating system binaries. This dramatically reduces the cost of designing and bringing up custom accelerators.

Further information on RISC-V is available from the nonprofit RISC-V Foundation, which was incorporated in August 2015 to help coordinate, standardize, protect, and promote the RISC-V ecosystem.[3] A free and open ISA is a critical ingredient in an agile hardware methodology, because having an active and diverse community of open-source contributors amortizes the overhead of maintaining and updating the software ecosystem, which makes it possible for smaller teams to focus on developing custom hardware.[4] A free ISA is also a prerequisite for shared open-source processor implementations, which can act as a base for further customization.[5]

Chisel

To facilitate agile hardware development by improving designer productivity, we developed Chisel[6] as a domain-specific extension to the Scala programming language. Chisel is not a high-level synthesis tool in which hardware is inferred from Scala code. Rather, Chisel is intended to be a substrate that provides a Scala abstraction of primitive hardware components, such as registers, muxes, and wires. Any Scala program whose execution generates a graph of such hardware components provides a blueprint to fabricate hardware designs; the Chisel compiler translates a graph of hardware components into a fast, cycle-accurate, bit-accurate C++ software simulator, or low-level synthesizable Verilog that maps to standard FPGA or ASIC flows.

Because Chisel is embedded in Scala, hardware developers can tap into Scala's modern



(a)

```
A class AHBXbar(nMasters: Int, nSlaves: Int) extends Module
B {
C      val io = new Bundle {
D        val masters = Vec(new AHBMasterIO, nMasters).flip
E        val slaves = Vec(new AHBSlaveIO, nSlaves).flip
F      }
G
H      val buses = List.fill(nMasters){Module(new AHBBus(nSlaves))}
I      val muxes = List.fill(nSlaves){Module(new AHBSlaveMux(nMasters))}
J
K      (buses.map(b => b.io.master) zip io.masters) foreach {
       case (b, m) => b <> m }
L      (muxes.map(m => m.io.out) zip io.slaves)  foreach {
       case (x, s) => x <> s }
M      for (m <- 0 until nMasters; s <- 0 until nSlaves) yield {
N        buses(m).io.slaves(s) <> muxes(s).io.ins(m)
O      }
P }
```

(b)

Figure 2. Using functional programming to write a parameterized AHB-Lite crossbar switch in Chisel. (a) Block diagrams. (b) Chisel source code. The code is short and easy to verify by inspection, but can support arbitrary numbers of masters and slave ports.

programming language features—such as object-oriented programming, functional programming, parameterized types, abstract data types, operator overloading, and type inference—to improve designer productivity by raising the abstraction level and increasing code reuse. Furthermore, metaprogramming, code generation, and hardware design tasks are all expressed in the same source language, which encourages developers to write parameterized hardware generators rather than discrete instances of individual blocks. This in turn improves code reuse within a given design and across generations of design iterations.

To see how these language features interact in real designs, we describe a crossbar that connects two AHB-Lite master ports to three AHB-Lite slave ports in Figure 2a. In the figure, the high-level block diagram is on the left, the two building blocks (AHB-Lite buses

```
   idiom                                                        result
A (1,2,3) map { n => n + 1 }                                    (2,3,4)
B (1,2,3) zip (a,b,c)                                           ((1,a),(2,b),(3,c))
C ((1,a),(2,b),(3,c)) map { case (left,right) => left }         (1,2,3)
D (1,2,3) foreach { n => print(n) }                             123
E for (x <- 1 to 3; y <- 1 to 3) yield (x,y)                    (1,1),(1,2),(1,3),(2,1),(2,2),(2,3),
                                                                (3,1),(3,2),(3,3)
```

Figure 3. Generic functional programming idioms to help understand the crossbar example.

and AHB-Lite slave muxes) are in the middle, and the final design is shown on the right. To help understand the Chisel code, Figure 3 shows some generic functional programming idioms. Idiom A maps a list of numbers to an expression that adds 1 to them. Idiom B zips two lists together. Idiom C iterates over a list of tuples, uses Scala's case statement to provide names of elements, and then operates on those names. Idiom D iterates over a list when the output values are not collected. Idiom E is a *for-comprehension* with a yield statement.

Figure 2b shows the Chisel code that builds the crossbar. Line A declares the AHBXbar module with two parameters: number of master ports (`nMasters`) and slave ports (`nSlaves`). Lines C through F declare the module's I/O ports. Lines H and I instantiate collections of AHB-Lite buses and AHB-Lite slave muxes. Lines K through N connect the building blocks together. Note that the Chisel operator <> connects module I/O ports together. Line K extracts the master I/O port of each bus, zips it to the corresponding master I/O port of the crossbar itself, and then connects them together. Line L does the same thing for each slave mux output and each slave I/O port of the crossbar. Lines M and N use for-comprehension to generate the cross-product of all port combinations, and use each pair to connect a specific slave I/O port of a specific bus to the matching input of the corresponding mux.

This code's brevity ensures that it is easily verified by inspection. Although the example shows two buses and three slave muxes, these 10 lines of code would be identical for any number of buses and slave muxes. In contrast, implementations of this crossbar module written in traditional HDLs, such as Verilog or VHDL, would be difficult to generalize into a parameterizable crossbar generator. This is just a small example of Chisel's power to bring modern programming language constructs to hardware development, increasing code maintainability, facilitating reuse, and enhancing designer productivity.

Rocket Chip Generator

Chisel's first-class support for object orientation and metaprogramming lets hardware designers write generators, rather than individual instances of designs, which in turn encourages the development of families of customizable designs. By encoding microarchitectural and domain knowledge in these generators, we can quickly create different chip instances customized for particular design goals and constraints.[7] Construction of hardware generators requires support for reconfiguring individual components based on the context in which they are deployed; thus, a particular focus with Chisel is to improve on the limited module parameterization facilities of traditional HDLs.

The Rocket chip generator is written in Chisel and constructs a RISC-V-based platform. The generator consists of a collection of parameterized chip-building libraries that we can use to generate different SoC variants. By standardizing the interfaces used to connect different libraries' generators to one another, we have created a plug-and-play environment in which it is trivial to swap out substantial components of the design simply by changing configuration files, leaving the hardware source code untouched. We can also test the output of individual generators and perform integration tests on the whole design, where the tests are also parameterized, so as to exercise the entire design under test.

Figure 4 presents the collection of library generators and their interfaces within the Rocket chip generator. These generators can be parameterized and composed into many various SoC designs. Their current capabilities include the following:

- *Core*: the Rocket scalar core generator with an optional floating-point unit, configurable pipelining of functional units, and customizable branch-prediction structures.
- *Caches*: a family of cache and translation look-aside buffer generators with configurable sizes, associativities, and replacement policies.
- *Rocket Custom Coprocessor*: the RoCC interface, a template for application-specific coprocessors that can expose their own parameters.
- *Tile*: a tile-generator template for cache-coherent tiles. The number and type of cores and accelerators are configurable, as is the organization of private caches.
- *TileLink*: a generator for networks of cache-coherent agents and the associated cache controllers. Configuration options include the number of tiles, coherence policy, presence of shared backing storage, and implementation of underlying physical networks.
- *Peripherals*: generators for AXI4-/AHB-Lite-/APB-compatible buses and various converters and controllers.

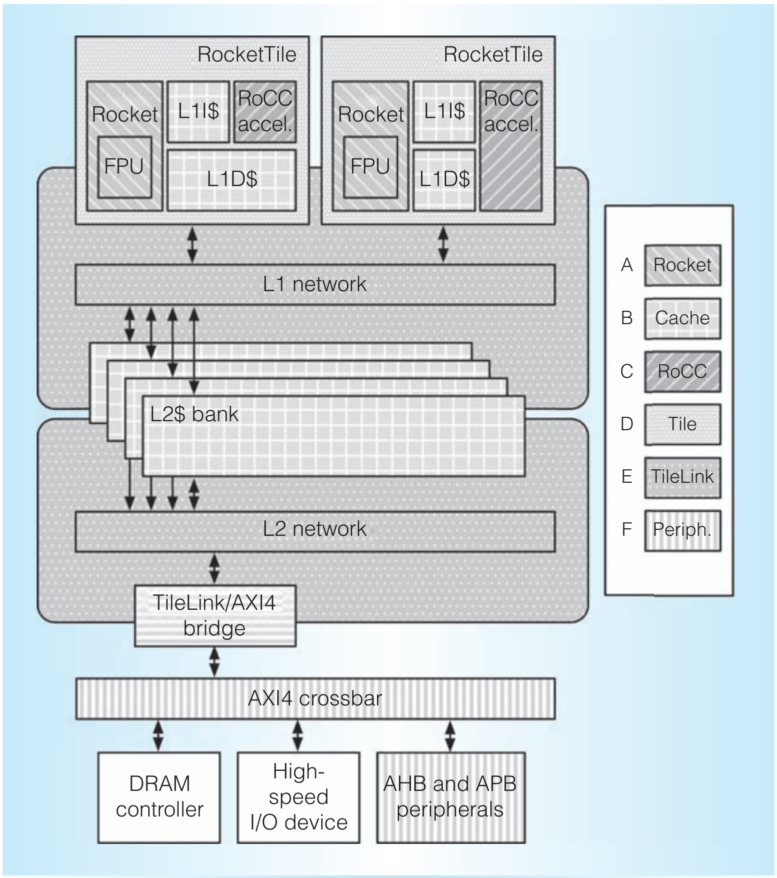


Figure 4. The Rocket chip generator comprises the following subcomponents: (a) a core generator, (b) cache generator, (c) Rocket Custom Coprocessor-compatible coprocessor generator, (d) tile generator, (e) TileLink generator, and (f) peripherals. These generators can be parameterized and composed into many various system-on-chip designs.

Physical Design Flow

We frequently translate Chisel's Verilog output into a physical design through an ASIC CAD toolflow. We have automated this flow and optimized it to generate industry-standard GDS-format mask files without designer intervention. Automatic nightly regressions provide the designer with regular feedback on cycle time, area, and energy consumption, and they also detect physical design problems early in the design cycle. Automatic verification and reporting on the quality of results allow more points in the design space to be evaluated, and reduce the barrier to implementing more significant changes at later stages in the development process.

One major hindrance to agile methodology for hardware design is CAD tool runtime. The deployment process for hardware is significantly more time-consuming than for software. Using an initial hardware description to generate a GDS that is electrically and logically correct can require many iterations. Because each iteration can take many hours to run to completion, the process could require weeks or months for larger designs. To avoid the productivity loss of these long-latency iterations, we initially focus on smaller designs, from which GDS can be generated more rapidly. Iterating on these smaller-scale tape-ins has proven valuable, because major problems are uncovered much earlier in the design process, and even more so when we target a new process technology. Additionally, the parameterizable nature of hardware generators reduces the

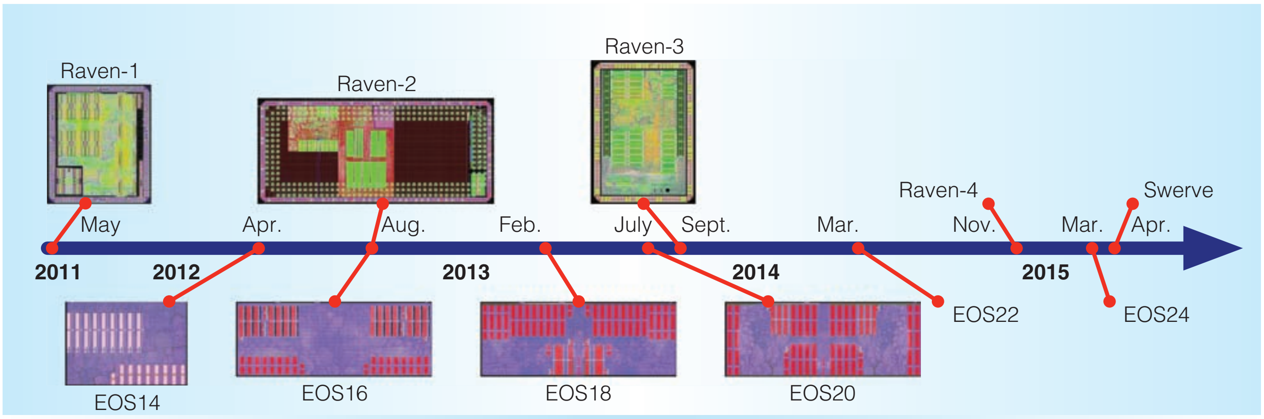


Figure 5. Recent UC Berkeley tape-outs using RISC-V processors for three lines of research. The Raven series investigated low-voltage reliable operation with on-chip DC–DC converters, the EOS series developed monolithically integrated silicon photonic links, and Swerve experimented with dynamic redundancy techniques to handle low-voltage faults.

effort to scale up to larger designs later in the design process. Regardless of design size, we ensure that each tape-in passes static timing analysis, is functionally equivalent to the RTL model, and has only a small number of design-rule violations. Together, these properties ensure that the design is indeed tape-out ready.

As a particular tape-out deadline approaches, we evaluate whether to tape out the most recently taped-in design, usually on the basis of whether the set of new features validated by the tape-in is sufficiently large. If we decide to tape out, we clean up the remaining design-rule violations and run final checks on the resulting GDS. The full cycle of taping out the GDS and testing the resulting chip is more involved than a tape-in, but qualitatively it is just another, longer, iteration in the agile methodology. Thus, we focus on producing lineages of increasingly complicated functioning chips, incorporating feedback from the previous chip into the next in the series. We believe that this rapid iterative process is essential to improve the productivity of hardware designers and enable iterative design-space exploration of alternative system microarchitectures.

Silicon Results

Figure 5 presents a timeline of UC Berkeley RISC-V microprocessor tape-outs that we created using earlier versions of the Rocket chip generator. The 28-nm Raven chips combine a 64-bit RISC-V vector microprocessor with on-chip switched-capacitor DC–DC converters and adaptive clocking. The 45-nm chips integrate a 64-bit dual-core RISC-V vector processor with monolithically integrated silicon photonic links.[8,9] In total, we taped out four Raven chips on STMicroelectronics' 28-nm fully depleted silicon-on-insulator (FD-SOI) process, six photonics chips on IBM's 45-nm SOI process, and one Swerve chip on TSMC's 28-nm process. The Rocket chip generator forms the shared code base for these 11 distinct systems; we incorporated the best ideas from each design back into the code base, ensuring maximal reuse even though the three distinct families of chips were specialized differently to test out distinct research ideas.

Case Study: RISC-V Vector Microprocessor with On-chip DC–DC Converters

We applied our agile design methodology to Raven-3, a 28-nm microprocessor that integrates switched-capacitor DC–DC converters, adaptive clocking circuitry, and low-voltage static RAM macros with a RISC-V vector processor instantiated by the Rocket chip generator.[10] The successful development of the Raven-3 prototype demonstrates how architecture and circuit research was supported by the combination of the freely extensible RISC-V ISA, the parameterizable Chisel-based hardware implementation, and

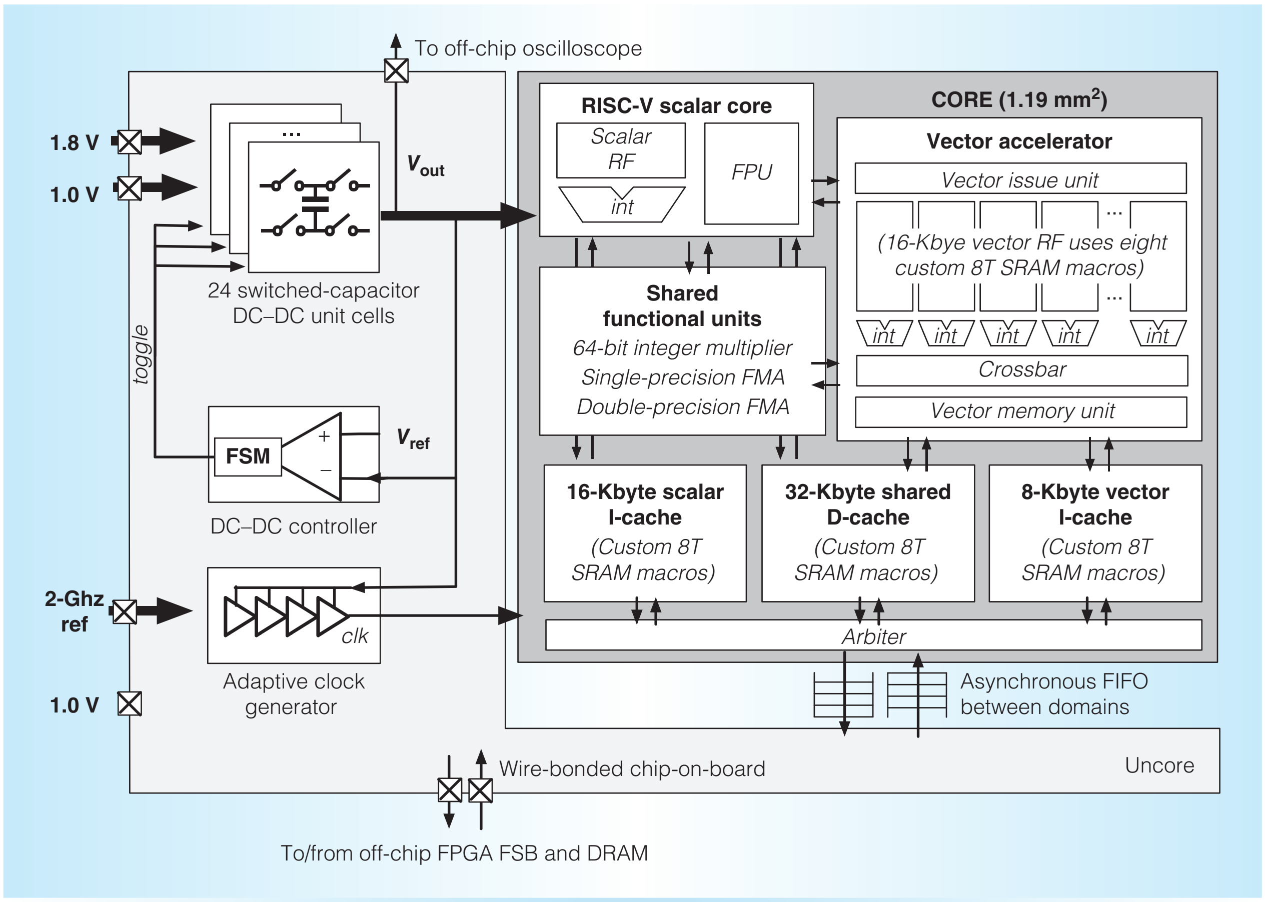


Figure 6. Raven-3 system-level block diagram. A RISC-V processor core with a vector coprocessor runs with varying voltage and frequency generated by on-chip switched-capacitor DC–DC converters and an adaptive clock generator.

fast feedback via iteration and automation as part of the agile methodology.

Figure 6 shows the Raven-3 system-level block diagram. The Raven-3 microprocessor instantiates the left RocketTile in Figure 4 with the Hwacha vector unit implemented as a RoCC accelerator. We did not include an L2 cache in this design iteration, leaving this feature for implementation in future tape-ins and tape-outs. The chip consists of two voltage domains: the varying core domain and the fixed uncore domain. The core domain connects to the output of fully integrated noninterleaved switched-capacitor DC–DC converters[11] and powers the processor and its L1 caches. The fixed 1-V domain powers the uncore. The DC–DC converters generate four voltage modes between 0.5 and 1 V from 1 and 1.8 V supplies; they achieve better than 80 percent conversion efficiency, while requiring no off-chip passives, which greatly simplifies package design.[10] The clock generator selects clock edges from a tunable replica circuit powered by the rippling core voltage that mimics the processor's critical path delay, and it adapts the clock frequency to the instantaneous voltage.[12]

The design of Raven-3 benefitted greatly from the agile hardware design methodology. By aspiring to build an incomplete prototype (rather than, for instance, a many-core design implementing these technologies), we dramatically improved the odds of project success. The unique integration of the power delivery, clocking, and processor systems was only possible because the design team broadly shared expertise about all aspects of the project. The processor RTL could largely be reused from previous tape-outs, with only incremental improvements and configuration

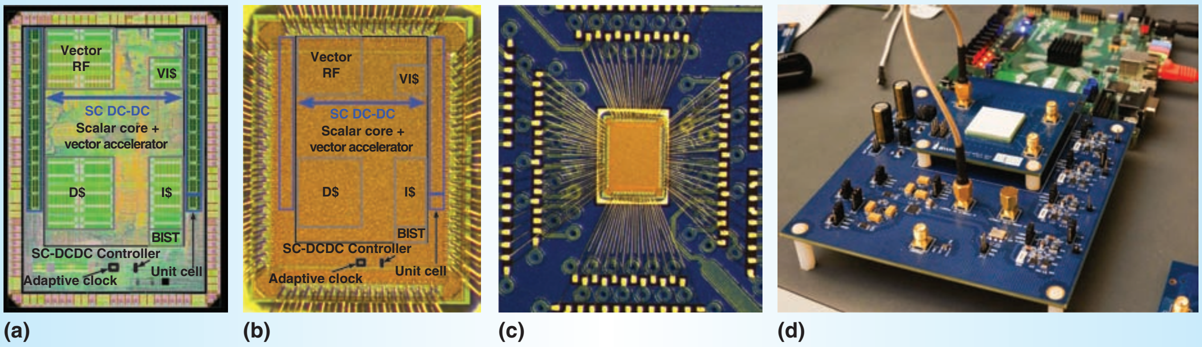


Figure 7. Raven-3 implementation: the (a) floorplan, (b) micrograph of the resulting chip, (c) wire-bonded chip on the package board, and (d) test setup.

changes required to validate it for the particular feature set desired. As physical design on partial feature sets gave incremental results, we were able to adjust design features to improve feasibility and QoR.

The agile hardware design methodology was especially important in this design due to the complexity and risk introduced by the custom voltage conversion and clocking, as illustrated by the following example. Tape-ins of a simple digital system with a single DC–DC unit cell were small enough to simulate at the transistor level, and these simulations found a critical bug in the DC–DC controller. These transistor-level simulations showed that a large current event could cause the output voltage to remain below the lower-bound reference voltage even after the converter switched. A bug in the controller would cause the converter to stop switching and drop the processor voltage to zero for this case. We believe this bug, which was fixed early in the design process by the addition of a simple state machine in the lower-bound controller, would have been found very late (if at all) with waterfall design approaches.

Figure 7 shows the floorplan and micrograph of the resulting chip, the wire-bonded chip on the package board, and the test setup. The resulting 2.37-mm^2 Raven chip is fully functional and boots Linux and executes compiled scalar and vector application code with the adaptive clock generator at all operating modes, while the DC–DC converter can transition between voltage modes in less than 20 ns. Raven-3 runs at a maximum clock frequency of 961 MHz at 1 V, consuming 173 mW, and at a minimum voltage of 0.45 V at 93 MHz, consuming 8 mW. Raven-3 achieves a maximum energy efficiency of 27 and 34 Gflops per watt while running a double-precision matrix-multiplication kernel on the vector accelerator with and without the switched-capacitor DC–DC converters, respectively. These results validate the agile approach to hardware design.

Our agile hardware development methodology played an important role in successfully taping out 11 RISC-V microprocessors in five years at UC Berkeley. With a combination of our agile methodology, Chisel, RISC-V, and the Rocket chip generator, a small team of graduate students at UC Berkeley was able to design multiple microprocessors that are competitive with industrial designs. Although there is much work left to do to show that this approach can scale up to the largest SoCs, we hope these projects serve as an opportunity to shed some light on inefficiencies in the current design process, to rethink how modern SoCs are built, and to revitalize hardware design. MICRO

Acknowledgments

We thank Tom Burd, James Dunn, Olivier Thomas, and Andrei Vladimirescu for their contributions and STMicroelectronics for the fabrication donation. This work was funded in part by BWRC, ASPIRE, DARPA PERFECT Award number HR0011-12-2-

0016, STARnet C-FAR, Intel ARO, AMD, SRC/TxACE, Marie Curie FP7, NSF GRFP, and an Nvidia graduate fellowship.

References

1. K. Beck et al., *The Agile Manifesto*, tech. report, Agile Alliance, 2001.
2. A. Waterman et al., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*, tech. report UCB/EECS-2014-54, EECS Dept., Univ. California, Berkeley, May 2014.
3. R. Merritt, "Google, HP, Oracle Join RISC-V," *EE Times*, 28 Dec. 2015.
4. V. Patil et al., "Out of Order Floating Point Coprocessor for RISC-V ISA," *Proc. 19th Int'l Symp. VLSI Design and Test*, 2015; doi:10.1109/ISVDAT.2015.7208116.
5. K. Asanović and D. Patterson, "The Case for Open Instruction Sets," *Microprocessor Report*, Aug. 2014.
6. J. Bachrach et al., "Chisel: Constructing Hardware in a Scala Embedded Language," *Proc. Design Automation Conf.*, 2012, pp. 1216–1225.
7. O. Shacham et al., "Rethinking Digital Design: Why Design Must Change," *IEEE Micro*, vol. 30, no. 6, 2010, pp. 9–24.
8. Y. Lee et al., "A 45nm 1.3GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators," *Proc. 40th European Solid-State Circuits Conf.*, 2014, pp. 199–202.
9. C. Sun et al., "Single-Chip Microprocessor that Communicates Directly Using Light," *Nature*, vol. 528, 2015, pp. 534–538.
10. B. Zimmer et al., "A RISC-V Vector Processor with Tightly-Integrated Switched-Capacitor DC–DC Converters in 28nm FDSOI," *Proc. IEEE Symp. VLSI Circuits*, 2015, pp. C316–C317.
11. H.-P. Le, S. Sanders, and E. Alon, "Design Techniques for Fully Integrated Switched-Capacitor DC–DC Converters," *IEEE J. Solid-State Circuits*, vol. 46, no. 9, 2011, pp. 2120–2131.
12. J. Kwak and B. Nikolić, "A 550-2260MHz Self-Adjustable Clock Generator in 28nm FDSOI," *Proc. IEEE Asian Solid-State Circuits Conf.*, 2015; doi:10.1109/ASSCC.2015.7387471.

Yunsup Lee is a PhD candidate in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received an MS in computer science from the University of California, Berkeley. Contact him at yunsup@eecs.berkeley.edu.

Andrew Waterman is a PhD candidate in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received an MS in computer science from the University of California, Berkeley. Contact him at waterman@eecs.berkeley.edu.

Henry Cook is a PhD candidate in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received an MS in computer science from the University of California, Berkeley. Contact him at hcook@eecs.berkeley.edu.

Brian Zimmer is a research scientist in the Circuits Research Group at Nvidia. He received a PhD in electrical engineering and computer science from the University of California, Berkeley, where he performed the research for this article. Contact him at brianzimmer@gmail.com.

Ben Keller is a PhD student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received an MS in electrical engineering from the University of California, Berkeley. Contact him at bkeller@eecs.berkeley.edu.

Alberto Puggelli is the director of technology at Lion Semiconductor. He received a PhD in electrical engineering from the University of California, Berkeley, where he performed the research for this article. Contact him at alberto.puggelli@gmail.com.

Jaehwa Kwak is a PhD candidate in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley. He received an MS in electrical engineering and computer sciences from Seoul National University. Contact him at jhkwak@eecs.berkeley.edu.

Ružica Jevtić is an assistant professor at the Universidad de Antonio de Nebrija in Madrid. She received a PhD in electrical engineering from Technical University of Madrid and was a postdoctoral fellow at the University of California, Berkeley, where she performed the research for this article. Contact her at rjevtic21@gmail.com.

Stevo Bailey is a PhD student in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley. He received an MS from the University of California, Berkeley. Contact him at stevo.bailey@eecs.berkeley.edu.

Milovan Blagojević is a PhD student in a CIFRE program at the Berkeley Wireless Research Center, STMicroelectronics, and Institut Supérieur d'Electronique de Paris. He received an MSc in electrical engineering from the University of Belgrade, Serbia. Contact him at milovan.blagojevic@st.com.

Pi-Feng Chiu is a PhD student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. She received an MS in electrical engineering from the National Tsing Hua University. Contact her at pfchiu@eecs.berkeley.edu.

Rimas Avižienis is a PhD candidate in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received an MS in computer science from the University of California, Berkeley. Contact him at rimas@eecs.berkeley.edu.

Brian Richards is a research staff engineer at the University of California, Berkeley, and a founding member of the Berkeley Wireless Research Center. He received an MS in electrical engineering and computer science from the University of California, Berkeley. Contact him at richards@eecs.berkeley.edu.

Jonathan Bachrach is an adjunct assistant professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received a PhD in computer science from the University of Massachusetts, Amherst. Contact him at jrb@pobox.com.

David Patterson is the Pardee Professor of Computer Science at the University of California, Berkeley. He received a PhD in computer science from the University of California, Los Angeles. Contact him at pattrsn@eecs.berkeley.edu.

Elad Alon is an associate professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley, and a codirector of the Berkeley Wireless Research Center. He received a PhD in electrical engineering from Stanford University. Contact him at elad@eecs.berkeley.edu.

Borivoje Nikolić is the National Semiconductor Distinguished Professor of Engineering at the University of California, Berkeley. He received a PhD in electrical and computer engineering from the University of California, Davis. Contact him at bora@eecs.berkeley.edu.

Krste Asanović is a professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He received a PhD in computer science from the University of California, Berkeley. Contact him at krste@berkeley.edu.