Theme Article

BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS

Christopher Celio, Pi-Feng Chiu, Krste Asanović, Borivoje Nikolić, and David Patterson
University of California, Berkeley

***Abstract*—The Berkeley resilient out-of-order machine (BROOM) is a resilient, wide-voltage-range implementation of an open-source out-of-order (OoO) RISC-V processor implemented in an ASIC flow. A 28-nm test-chip contains a BOOM OoO core and a 1-MiB level-2 (L2) cache, enhanced with architectural error tolerance for low-voltage operation. It was implemented by using an agile design methodology, where the initial OoO architecture was transformed to perform well in a high-performance, low-leakage CMOS process, informed by synthesis, place, and route data by using foundry-provided standard-cell library and memory compiler. The two-person-team productivity was improved in part thanks to a number of open-source artifacts: The *Chisel* hardware construction language, the RISC-V instruction set architecture, the rocket-chip SoC generator, and the open-source BOOM core. The resulting chip, taped out using TSMC's 28-nm HPM process, runs at 1.0 GHz at 0.9 V, and is able to operate down to 0.47 V.**

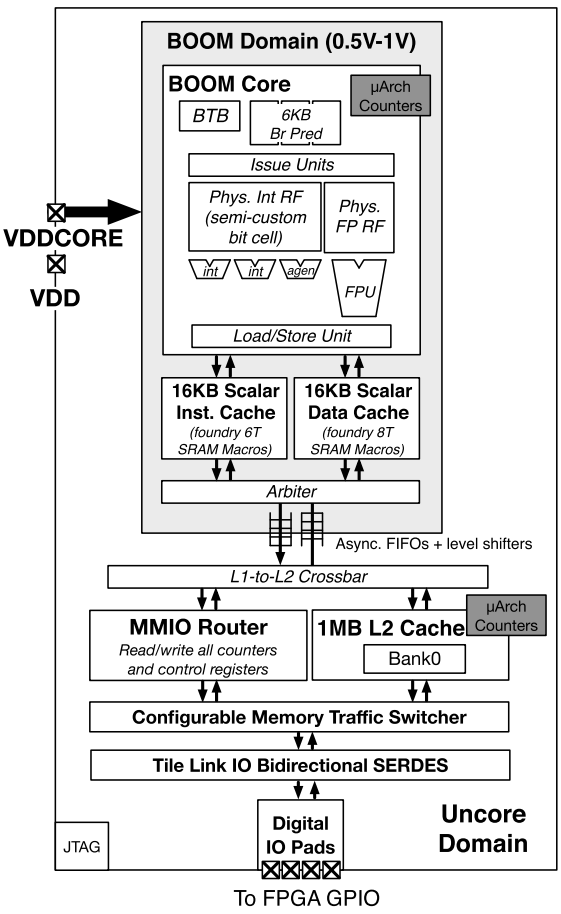
RISC-V is an open-source instruction set architecture (ISA) that is gaining wide attention. There are several open-source and commercial in-order cores that implement the RISC-V ISA; however, there is a need for high-performance cores. BOOM is a synthesizable, parameterized, superscalar out-of-order (OoO) RISC-V core, that has been originally designed to serve as the prototypical baseline processor for future microarchitectural studies of OoO processors. Its original goal was to provide a readable, open-source implementation for use in education, research, and industry, and had been

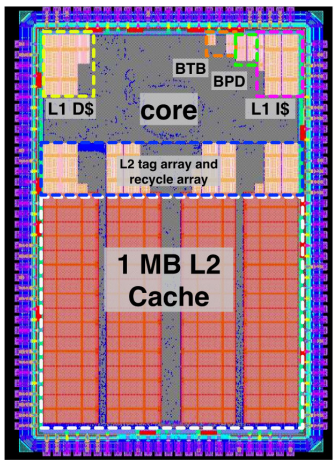
Digital Object Identifier 10.1109/MM.2019.2897782
Date of publication 5 February 2019; date of current version 15 March 2019.

Published by the IEEE Computer Society

evaluated through educational standard-cell libraries. The Berkeley Resilient OoO Machine (BROOM) contains an evolved version of BOOM: the core has been transformed to explore the design space in a representative process for high-performance mobile applications. It has been designed in an ASIC flow, which enabled a rapid evaluation of changes to the RTL and physical design to improve the performance of the processor. Figure 1 shows the block diagram of the *BROOM* processor. *BROOM* consists of a single BOOM core and a 1-MiB L2 cache, each in their own clock and voltage domains.

The additional feature of the test chip is the architectural resiliency techniques for operation of the cache in a wide voltage range, enabling the processor to operate with a high efficiency at low voltages.





ISA	RISC-V RV64IMAFD
BTB	64 x 4 (set-associative)
BPD	6 kB gshare
Fetch Width	2 instructions
Issue Width	4 micro-ops
Issue Entries	16 (int)/16(mem)/16 (fp)
Regfile	6R3W (Int) 3R2W (FP)
Frequency	1.0 GHz at 0.9 V
Area	4.86 mm^2 72 M transistors
Coremark Performance	3.77 CM/MHz 1.11 IPC

Figure 1. *BROOM* chip block diagram, annotated place-and-routed chip plot, and feature summary.

BROOM was implemented using LVT-based standard cells and a foundry-provided memory compiler. The entire chip measures less than 2 mm × 3 mm and is composed of 72 million transistors. The chip is composed of 417 000 standard cells and 73 SRAM macros; the core and L1 caches make up 310 000 cells and 20 SRAM macros. The final sign-off in the slow-slow corner was at 1.68 ns. Figure 1 shows the placed-and-routed chip plot.

LEVERAGING OPEN-SOURCE INFRASTRUCTURE

BOOM implements the open-source RISC-V ISA, which was designed from the ground-up to enable technology-driven computer architecture research. The clean and simple design of RISC-V allows for a focus on the processor without getting weighed down with awkward instructions that demand undue attention or spending extra effort managing software ports.

BOOM is written in *Chisel*, an open-source hardware construction language developed to enable the advanced hardware design. *Chisel* allows designers to utilize concepts such as object orientation, functional programming, parameterized types, and type inference which makes it easier to implement highly parameterized hardware generators. However, *Chisel* is *not* a high-level synthesis language—the primitives provided by *Chisel* are, for example, registers, wires, and memories. One of *Chisel*'s strengths is its focus on generating well formed, synthesizable Verilog. This feature decreased design risk. *Chisel* also brings software development-level productivity to the RTL coding, and helps encourage focusing implementation efforts on writing generators, rather

than a single design instance. For example, the open-source RISC-V *Rocket-chip* generator presents a template for designing systems-on-a-chip (SoCs). *Rocket-chip* supports coherent multilevel caches and standard interconnects. BOOM makes significant use of *Rocket-chip* as a library—the caches, the uncore, and the functional units all derive from *Rocket*. In total, over 11 500 lines of code (LOC) are instantiated by BOOM from the *Rocket-chip* repository.

BOOM CORE

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

As BOOM is a synthesizable processor, we must rely on microarchitecture-level techniques to address critical paths and add more pipeline stages to trade off instructions per cycle (IPC), cycle time (frequency), and design complexity. However, as process nodes have become smaller, transistor leakage and variability has increased, and power efficiency restrictive, many of the more aggressive custom techniques have become more difficult and expensive to apply.[4] Modern high-performance processors have largely limited their custom design efforts to more regular structures such as memories and register files.

We began our design efforts with *BOOMv1*; a version of BOOM whose implementation was informed using educational technology libraries and CACTI cache models. BOOMv1 follows the 6-stage pipeline structure of the MIPS R10K—*fetch*, *decode/rename*, *issue/register-read*, *execute*, *memory*, and *writeback*. For design simplicity, all uops are placed into a single unified issue window. Likewise, all physical registers (both integer and floating-point registers) are located in a single unified physical register file. BOOMv1 also utilized a short 2-stage front-end pipeline. Conditional branch prediction occurs after the branches have been decoded.

The design of BOOMv1 was partly informed by using educational technology libraries in conjunction with synthesis-only tools. BOOMv1 used Cacti[5] to analytically model the characteristics of memories, which is oriented toward the single-port, cache-sized SRAMs. However, BOOM makes use of a multitude of smaller, irregular SRAMs for modules such as branch predictor (BPD) tables, and address target buffers. Figure 2 lists all of the SRAM macros used within the BOOM core.

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

1) issue window select;
2) register rename busy-table read;
3) conditional BPD redirect;
4) register file read.

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

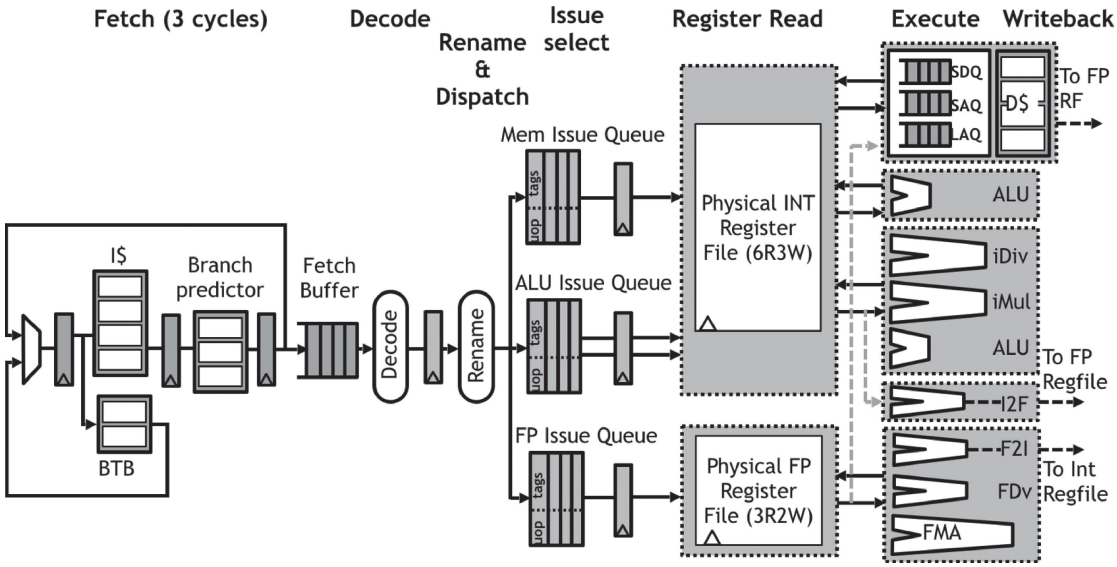
BOOMv2: IMPROVING BOOM's QUALITY-OF-RESULTS

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

Migration to BOOMv2 included 4948 additions and 2377 deleted LOC out of the total 16 000 LOC code base. The following sections describe some of the major changes that comprise the BOOMv2 update.

Frontend (Instruction Fetch)

Processor performance is best when the frontend provides an uninterrupted stream of instructions. This requires the frontend to utilize branch prediction techniques to predict which path it believes the instruction stream will take



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

Figure 2. Final BOOM core configuration used in the *BROOM* chip, as well as the configurations used for each of the SRAM macros used within the BOOM core.

long before the branch can be properly resolved. A number of different predictors are used, each trading off accuracy, area, critical path cost, and pipeline penalty when making a prediction.

The *Branch Target Buffer (BTB)* maintains a set of tables mapping from *instruction addresses* to *branch targets*. Some *hysteresis* bits are used to help guide the *taken/not-taken* decision of the BTB in the case of a *tag hit*. The BTB is a very expensive structure—each BTB entry contains a *tag* and a *target*. The BTB also contains a *return address stack* for predicting the function returns.

To improve a critical path and increase the capacity, we replaced BOOMv1's fully tagged, fully associative BTB design with a partially tagged, set-associative BTB. We also implemented the new BTB using single-ported SRAM macros, instead of flip-flops.

The *Conditional BPD* maintains a set of prediction and hysteresis tables to make *taken/not-taken* predictions based on a *look-up address*. The BPD *only* makes *taken/not-taken* predictions—it therefore relies on some other agent to provide information on what instructions are branches and what their targets are. The BPD can either use the BTB for this information or it can wait and decode the instructions themselves. Because the BPD does not store the branch targets, it can be much denser and more accurate than the BTB.

BOOM uses a global history predictor, which works by tracking the outcome of the last N branches in the program and hashes this *history* with the *look-up address* to compute an index into the prediction tables. BOOM's predictor tables are implemented using single-ported SRAMs. Although many prediction tables are conceptually "tall and skinny" matrices (thousands of 2- or 4-bit entries), a generator written in *Chisel* transforms the predictor tables into a square memory structure to best match the SRAMs provided by a memory compiler.

We found a critical path in BOOMv1 to be the BPD making a prediction and redirecting the fetch instruction address, as the BPD must first decode

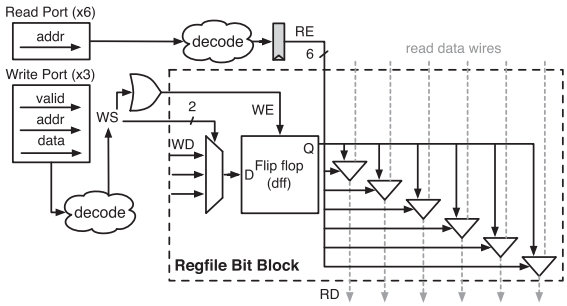


Figure 3. Register File Bit manually crafted out of foundry-provided standard cells. Each read port provides a read-enable bit to signal a tri-state buffer to drive its port's read data line. The register file bits are laid out in an array for placement with guidance to the place tools. The tools are then allowed to automatically route the 18 wires into and out of each bit block.

the newly fetched instructions and compute potential branch targets before it can redirect fetch. For BOOMv2, we moved the BPD array access back a stage to now operate in parallel with decoding the instructions. The final prediction and redirection are then performed at the beginning of the following stage (see Figure 2). Moving the BPD redirection back a cycle also gave us the freedom to provide a full cycle for the hash indexing function, which removes the hashing off the critical path of *Next-PC selection*. However, pushing back the BPD redirection a stage comes at the cost of an extra bubble on BPD redirections.

Distributed Issue Windows

The *issue window* holds all inflight and unexecuted micro-ops (uops). For BOOM, the issue window is implemented as a collapsing queue to allow the oldest instructions to be compressed toward the top. For issue-select, a cascading priority encoder selects the oldest instruction that is ready to issue. This path is exacerbated either by increasing the number of entries or by increasing the number of issue ports. For BOOMv1, our synthesizable implementation of a 20-entry issue window with three issue ports was found to be too aggressive, so we switched to three distributed issue windows with 16 entries each (separate windows for integer, memory, and floating-point operations). This removes issue-select from the critical path while also increasing the total number of instructions that can be scheduled. However, to maintain performance of executing two integer ALU instructions and one memory instruction per cycle, a common configuration of BOOM uses two issue-select ports on the integer issue window.

Custom Bit-Array Register File Design

One of the critical components of an OoO processor, and most challenging to synthesize in a standard ASIC flow, is the multiported register file. BOOM's register file required both microarchitectural adjustments and a semicustom physical design to achieve the desired performance. The design of a register file provides many challenges—reading data out of the register file is a critical path, and routing read data to functional units is a routing challenge. Both the number of registers and the number of ports further exacerbate the challenges of synthesizing the register file.

The first path to improving the register file design was purely microarchitectural. The *issue-select* and *register-read* stages were split into two separate stages—each now gets a full cycle to themselves. The register count is lowered by splitting up the unified physical register file into separate floating-point and integer register files. This split also allows for reducing the read-port count by moving the three-operand fused-multiply add floating-point unit to the smaller floating-point register file.

The second path to improving the register file involved physical design. A significant problem in placing and routing a register file is the issue in routing many wires to a relatively dense regfile array. BOOMv2's 70 entry integer register file of six read ports and three write ports comes to 4480 bits, each needing 18 wires routed into and out of it. There is a mismatch between the synthesized array and the area needed to route all required wires, resulting in routing congestion.

The register file in this design was implemented by semicustom crafting a register file *bit* out of foundry-provided standard cells (see Figure 3). The *Chisel* register file was blackboxed, and a lower level of hierarchy was

manually described in structural Verilog in which standard cells were instantiated to construct a bit-cell with its access ports. The bit-cells were preplaced and the router automatically routed wires correctly to complete the register file.

Although the register file bits are implemented in a structural Verilog, the decode logic and peripheral circuitry are implemented in *Chisel*. We also implemented a behavioral model of the custom array in *Chisel* to verify the decode logic through RTL simulation and then performed additional verification of the custom bit-array register file in gate-level simulation.

To support the target cycle time, the register file is implemented by using *hierarchical bitlines*; the bits are divided into clusters, tristates drive the read ports inside of each cluster, and muxes select the read data across clusters. This prevents the tristate buffers from having to drive each read wire across all 70 registers.

As a counterpoint, the smaller floating-point register file (three read ports, two write ports) is fully synthesized with no placement guidance. Aside from the integer register file and the SRAMs, no other logic in *Chisel* was implemented via Verilog blackboxes.

MICROARCHITECTURAL ASSIST TECHNIQUES

Low-voltage operation improves energy efficiency. Unfortunately, SRAM-based memories tend to fail first as voltage is lowered, suffering as much as an order of magnitude (10×) increase in bit errors for every 50 mV reduction in Vdd. To enable low-voltage operation, we implemented a number of features that allows the processor to *tolerate* higher error rates. All of these techniques were implemented at the RTL-level in *Chisel*:

1) line disabling (LD);
2) line recycling (LR);
3) dynamic column redundancy (DCR);
4) bit bypass with SRAM (BB-S) for tag protection.

A built-in self-test checks for erroneous bits at boot-time after the voltage has been set. SRAM-based cache lines with bad bits can be disabled (LD). LD is a common technique, but it reduces capacity. Some of the capacity can be recovered by using line recycling—three disabled lines can be aggregated via majority-vote to regain 33% of the disabled line capacity, so long as each line's failing bit is in a different location. LR was only used to protect the L2 cache data arrays.

Dynamic column redundancy (DCR) adds an extra bit to each cache set, and uses a multiplexer shift driven by a Redundancy Address to dynamically avoid the erroneous bit. Finally, the Bit Bypass with SRAM (BB-S) technique focuses on protecting erroneous bits in the tag arrays. Bit bypass uses flip-flops to store the necessary repair bits to fix a limited number of bad entries. Our BB-S scheme stores the repair error location information in SRAM for every tag entry, saving on area and reducing the BB tag search to find potential matching entries.

The details of the resilient cache design, including the measurement results, are discussed in more detail.[6]

AGILE DESIGN APPROACH

Figure 4 shows all chip builds and their critical path lengths performed over a four-month period as part of the *BROOM* tapeout effort. This involved the microarchitectural transformation of BOOMv1 to BOOMv2, and physical design of the chip. Data from postsynthesis ("syn") and post-place-and-route ("par") are shown and include builds performed at both the slow-slow (SS) typical-typical (TT) corners. For our flow, the SS corner was 0.81 V at 125 C and the TT corner was 0.9 V at 25 C. Early builds were only of a BOOM core plus an L2 cache while later builds add in the resiliency ("res") hardware. One should be careful of drawing conclusions from this figure; most builds resulted in LVS and DRC violations and many changes were made between each build. For example, early builds explored shrinking structure sizes to find the most fundamental critical paths while later builds sought to find the upper limits of structure sizing before the post-place-and-route critical path noticeably worsened.

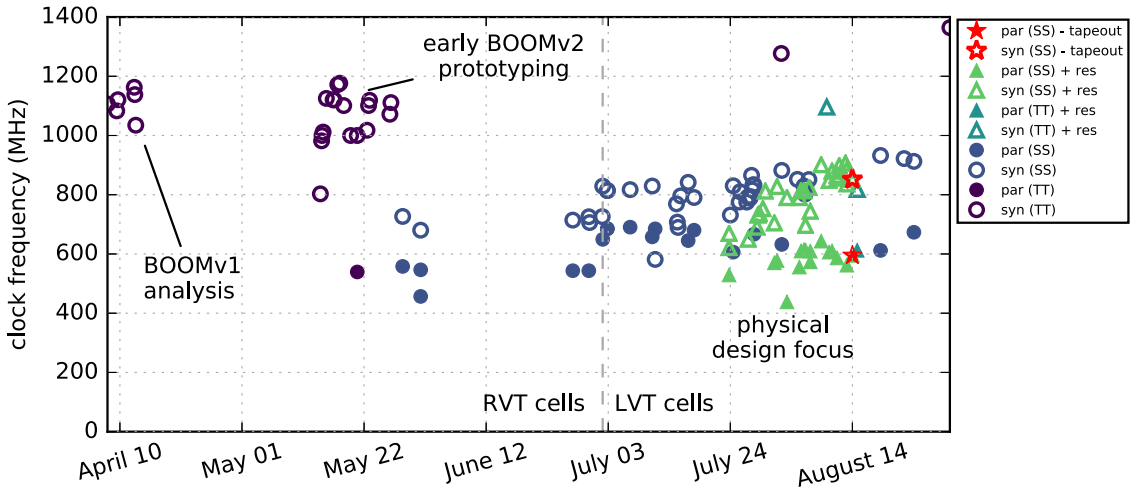


Figure 4. All VLSI builds are shown by date. Both slow-slow (SS) and typical-typical (TT) corners are shown. RVT cells were used initially, but replaced with LVT cells starting in July. In the last month of the implementation effort, we added in the resiliency hardware (*"res"*) central to the research thesis of the chip which added to the critical path. While our design efforts slowly improved the postsynthesis critical paths, post-place-and-route reports showed the clock frequency was less amenable to our efforts. Not shown is the impact of our design efforts on removing any LVS and DRC errors. Thus, many of the builds do *not* represent a manufacturable design.

The *BROOM* tapeout effort started with a preliminary analysis of the BOOM's quality-of-result (QoR). This effort was performed using RVT-based cells and targeting the TT corner. By changing BOOM's configurations, we could build an intuition of what critical paths were truly critical and arrive at a plan of action for addressing these paths with a mixture of microarchitectural changes and physical design effort. For example, by removing an execution unit or shrinking the issue window size, we could better understand the benefits of design changes that would provide fewer issue ports per issue window. At this stage, we had concluded that four critical paths needed to be managed. As previously mentioned in Section 3, these critical paths were as follows:

1) issue window select;
2) register rename busy-table read;
3) conditional BPD redirect;
4) register file read.

The microarchitectural changes to address the first two items together took one month. We also quickly prototyped a new frontend design that approximated a critical path fix for item three but was otherwise functionally incorrect. This frontend prototype helped justify the necessary design work before we committed to a full redesign of the frontend. We began testing these new changes in mid-May and labeled the new design *BOOMv2*. Figure 4 shows the cluster of activity that correspond to the BOOMv1 and early BOOMv2 analysis. After the initial BOOMv2 analysis was performed, another month of design effort went into BOOM to finish implementing the new frontend design and to apply changes based on the initial performance feedback.

Half-way through the design cycle (two months into the effort), as the BOOMv2 RTL effort was wrapping up, the implementation focus switched to physical design. Parameters in BOOM, for example, the ROB size or the BPD sizing, were reduced to get a better feel for the fundamental critical paths that still required work and to find which modules had the greatest effect on DRC and LVS errors. At this stage, the clock frequency improved as the BOOM parameters were changed to instantiate a smaller BOOM core.

Once the BOOM microarchitecture was settled, we added the resiliency hardware to the design. Some of these resiliency structures are on the critical paths of SRAM accesses. Thus, any chip builds with resiliency hardware enabled may generate analysis reports that hide critical paths

that still need attention in the BOOM RTL. To allow improvements to both the resiliency structures and to the BOOM core to occur in parallel, we continued to perform chip builds with and without the resiliency hardware enabled.

As our attention shifted to physical design issues, the major issue was the design of a 6-read, 3-write register file. Semicustom design was chosen over placement hints to the tools, for better QoR and faster design convergence.

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

BOOM is an open-source OoO superscalar RISC-V processor that can be used for architecture exploration, and education, but also in practical industrial designs.

Over the course of our tape-out effort, the syn results slowly improved. This was aided by our RTL productivity and our 2–3-h synthesis runs. However, par results proved more stubborn and stayed mostly flat. Congested designs took 16 h to route, giving us less time to iterate, and changes in placements resulted in unintuitive changes in the resulting critical paths. Alas, most par effort was focused on fixing DRC and LVS issues and not on fixing timing.

The design was taped out after the four-month design cycle. As the effort of this design project was to explore the superscalar processor design in an ASIC flow, we continued making changes to the RTL to improve the QoR. Each additional build continued to provide us new critical paths to address. The final critical path of the place and routed design was through the resiliency error logging code. Our final sign-off at the SS corner was 1.17 ns after synthesis and 1.68 ns after place-and-route. The resulting chip was demonstrated to run up 1.0 GHz at the nominal 0.9 V and down to 0.47 V at 70 MHz with assist techniques. Without assistance, *BROOM* was able to operate down to 0.6 V.

The design that has been fabricated is not the ultimate BOOM design, as both its clock frequency and the IPC performance can be improved. The measured Coremark performance was 3.77 CM/MHz with 1.11 IPC, limited by the branch prediction accuracy and the long load-to-use delay introduced while fixing timing paths. While some issues have since been addressed, such as the addition of load-cache-hit speculation bypassing to improve load-to-use, other improvements are ongoing. Future VLSI implementation efforts can continue from a known, good design point and can build on the early exploratory builds that were needed for the *BROOM* tapeout.

CONCLUSION

BOOM is an open-source OoO superscalar RISC-V processor that can be used for architecture exploration, and education, but also in practical industrial designs. Modern OoO processors rely on a number of memory macros and arrays of different shapes and sizes, and many of them appear in the critical path when designed in a standard ASIC flow. The impact on the actual critical path is hard to assess by using flip-flop-based arrays and academic/educational modeling tools, because they may either yield physically unimplementable designs or generate designs with poor performance and power characteristics. Rearchitecting the design by relying on a hand-crafted, yet synthesizable register file array and leveraging hardware generators written in *Chisel* helped us isolate real critical paths from false ones. This methodology narrows down the range of arrays that would eventually have to be handcrafted for a serious production-quality implementation. Describing hardware using generators also helped us explore multiple design points, with the final design choices being committed to later in the design cycle.

Chisel is a highly expressive language. With a proper software engineering of the code base, radical changes to the data-paths can be made very quickly. However, physical design is often a stumbling block to agile hardware development. Small changes could be reasoned about and executed swiftly, but larger changes could change the physical layout of the chip and dramatically

affect critical paths and the associated costs of the new design point.

The BOOM core is still being developed and we can expect further refinements.

REFERENCES

[1] K. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 39, no. 2, pp. 28–41, Apr. 1996.

[2] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar./Apr. 1999.

[3] D. G. Chinnery *et al.*, "Closing the power gap between asic and custom: An asic perspective," in *Proc. 42nd Annu. Des. Autom. Conf.*, 2005, pp. 275–280.

[4] M. Anderson, "A more cerebral cortex," *IEEE Spectrum*, vol. 47, no. 1, pp. 58–63, Jan. 2010.

[5] S. J. Wilton *et al.*, "CACTI: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[6] P.-F. Chiu *et al.*, "Cache resiliency techniques for low-voltage RISC-V out-of-order processor in 28 nm CMOS," submitted to *IEEE Solid-State Circuits Letters*, 2019, DOI: 10.1109/LSSC.2019.2900148.

Christopher Celio is a CPU architect at Esperanto Technologies. He received the Ph.D. degree in computer science from the University of California, Berkeley, CA, USA, where he performed research for this article. Contact him at celio@eecs.berkeley.edu.

Pi-Feng Chiu is a Technologist in Memory and Networking Fabrics Research at Western Digital. She received the Ph.D. degree in electrical engineering from the University of California, Berkeley, CA, USA, where she performed research for this article. Contact her at pfchiu@eecs.berkeley.edu.

Krste Asanović is a Professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, USA. He received the Ph.D. degree in computer science from the University of California, Berkeley. He is a Fellow of IEEE. Contact him at krste@berkeley.edu.

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

David Patterson is the Pardee Professor of Computer Science, Emeritus at the University of California, Berkeley, CA, USA. He received the Ph.D. degree in computer science from the University of California, Los Angeles, CA, USA. He is a Fellow of IEEE. Contact him at pattrsn@eecs.berkeley.edu.