BOOM v2: an open-source out-of-order RISC-V core

Christopher Celio
Pi-Feng Chiu
Borivoje Nikolic
David A. Patterson
Krste Asanović

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-157
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

September 26, 2017

Acknowledgement

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

BOOM v2

an open-source out-of-order RISC-V core

Christopher Celio, Pi-Feng Chiu, Borivoje Nikolić, David Patterson, Krste Asanović
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
celio@eecs.berkeley.edu

ABSTRACT

This paper presents BOOM version 2, an updated version of the Berkeley Out-of-Order Machine first presented in [3]. The design exploration was performed through synthesis, place and route using the foundry-provided standard-cell library and the memory compiler in the TSMC 28 nm HPM process (high performance mobile).

BOOM is an open-source processor that implements the RV64G RISC-V Instruction Set Architecture (ISA). Like most contemporary high-performance cores, BOOM is superscalar (able to execute multiple instructions per cycle) and out-of-order (able to execute instructions as their dependencies are resolved and not restricted to their program order). BOOM is implemented as a parameterizable generator written using the Chisel hardware construction language [2] that can used to generate synthesizable implementations targeting both FPGAs and ASICs.

BOOMv2 is an update in which the design effort has been informed by analysis of synthesized, placed and routed data provided by a contemporary industrial tool flow. We also had access to standard single- and dual-ported memory compilers provided by the foundry, allowing us to explore design trade-offs using different SRAM memories and comparing against synthesized flip-flop arrays. The main distinguishing features of BOOMv2 include an updated 3-stage front-end design with a bigger set-associative Branch Target Buffer (BTB); a pipelined *register rename* stage; split floating point and integer register files; a dedicated floating point pipeline; separate issue windows for floating point, integer, and memory micro-operations; and separate stages for *issue-select* and *register read*.

Managing the complexity of the register file was the largest obstacle to improving BOOM's clock frequency. We spent considerable effort on placing-and-routing a semi-custom 9-port register file to explore the potential improvements over a fully synthesized design, in conjunction with microarchitectural techniques to reduce the size and port count of the register file. BOOMv2 has a 37 fanout-of-four (FO4) inverter delay after synthesis and 50 FO4 after place-and-route, a 24% reduction from BOOMv1's 65 FO4 after place-and-route. Unfortunately, instruction per cycle (IPC) performance drops up to 20%, mostly due to the extra latency between load instructions and dependent instructions. However, the new BOOMv2 physical design paves the way for IPC recovery later.

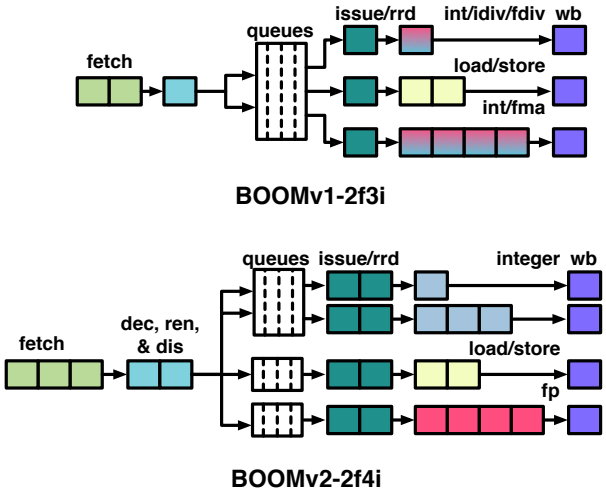


Figure 1: A comparison of a three-issue (*3i*) BOOMv1 and four-issue (*4i*) BOOMv2 pipeline both which can fetch two instructions every cycle (*2f*). Both show two integer ALU units, one memory unit, and one floating point unit. Note that BOOMv2 uses a distributed issue window to reduce the issue port count for each separate issue window. In BOOMv1 the floating point unit shares an issue port and register access ports with an integer ALU. Also note that BOOMv1 and BOOMv2 are parameterizable, allowing for wider issue widths than shown here.

1 INTRODUCTION

BOOM was inspired initially by the MIPS R10K and Alpha 21264 processors from the 1990s, whose designs teams provided relatively detailed insight into their processors' microarchitectures [6, 7, 11]. However, both processors relied on custom, dynamic logic which allowed them to achieve very high clock frequencies despite their very short pipelines[1] – the Alpha 21264 has 15 fanout-of-four (FO4)[2] inverter delays [4].

[1] The R10K has five stages from instruction fetch to integer instruction write-back and the Alpha 21264 has seven stages for the same path. Load instructions take an additional cycle for both processors.
[2] An inverter driving four times its input capacitance. FO4 is a useful, relatively technology-agnostic measurement of a circuit path length.

CARRV 2017, October 2017, Boston, Massachusetts USA
2017.

Table 1: The parameters chosen for analysis of BOOM. Although BOOM is a parameterizable generator, for simplicity of discussion, we have limited our analysis to these two instantiations.

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

As a comparison, the synthesizable[3] Tensilica's Xtensa processor, fabricated in a 0.25 micron ASIC process and contemporary with the Alpha 21264, was estimated to have roughly 44 FO4 delays [4].

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. The exploration in this work is performed by using only the available single-and dual-ported memory compilers, without custom circuit design. It should be noted that, as process nodes have become smaller, transistor variability has increased, and power-efficiency has become restricting, many of the more aggressive custom techniques have become more difficult and expensive to apply [1]. Modern high-performance processors have largely limited their custom design efforts to more regular structures such as arrays and register files.

2 BOOMV1

BOOMv1 follows the 6-stage pipeline structure of the MIPS R10K – *fetch*, *decode/rename*, *issue/register-read*, *execute*, *memory*, and *writeback*. During *decode*, instructions are mapped to *micro-operations* (uops) and during *rename* all logical register specifiers are mapped to physical register specifiers. For design simplicity, all uops are placed into a single unified issue window. Likewise, all physical registers (both integer and floating point registers) are located in a single unified physical register file. Execution Units can contain a mix of integer units and floating point units. This greatly simplifies floating point memory instructions and floating point-integer conversion instructions as they can read their mix of integer and floating point operands from the same physical register file. BOOMv1 also utilized a short 2-stage front-end pipeline design. Conditional branch prediction occurs after the branches have been decoded.

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

The design of BOOMv1 was partly informed by using educational technology libraries in conjunction with synthesis tools. While using educational libraries was useful for finding egregious mistakes in control logic signals, it was less useful in informing the organization of the datapaths. Most critically, we lacked access to a commercial memory compiler. Although tools such as Cacti [10] can be used to analytically model the characteristics of memories, Cacti works best for reproducing memories that it has been tuned against such as single-port, cache-sized SRAMs. However, BOOM makes use of a multitude of smaller SRAM arrays for modules such as branch predictor tables, prediction snapshots, and address target buffers.

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

(1) issue select
(2) register rename busy table read
(3) conditional branch predictor redirect
(4) register file read

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

3 BOOMV2

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

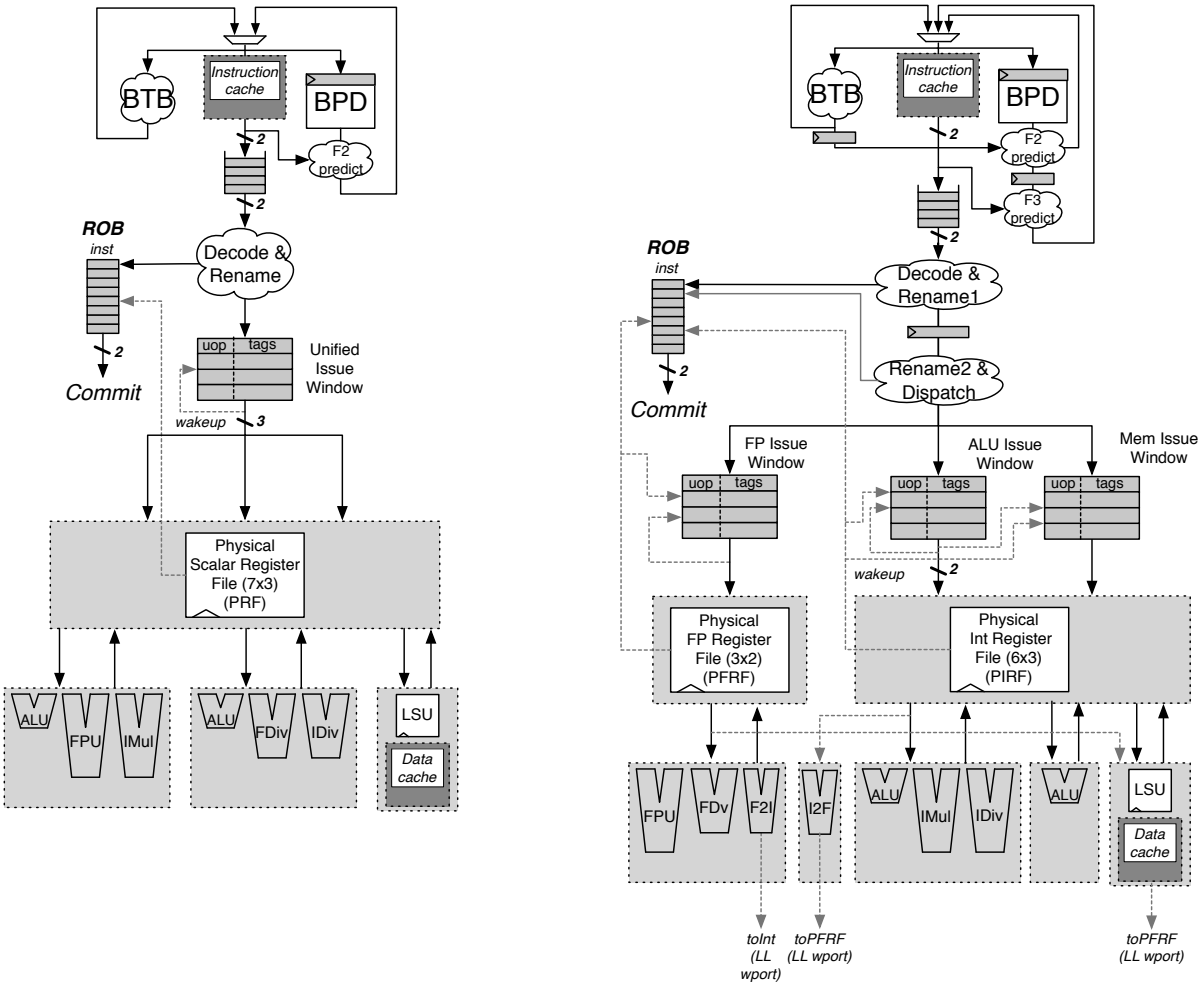
Work on BOOMv2 took place from April 9th through Aug 9th and included 4,948 additions and 2,377 deleted lines of code (LOC) out of the now 16k LOC code base. The following sections describe some of the major changes that comprise the BOOMv2 design.

3.1 Frontend (Instruction Fetch)

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

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

Branch Target Buffer (BTB) The BTB maintains a set of tables mapping from *instruction addresses* (PCs) to *branch targets*. When a lookup is performed, the *look-up address* indexes into the BTB and looks for any *tag matches*. If there is a *tag hit*, the BTB will make a prediction and may redirect



(a) BOOMv1 **(b) BOOMv2**

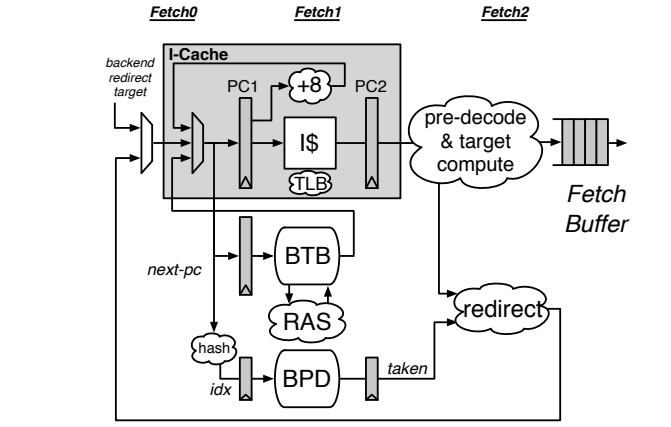
Figure 2: The datapath changes between BOOMv1 and BOOMv2. Most notably, the issue window and physical register file have been distributed and an additional cycle has been added to the *Fetch* and *Rename* stages.

the frontend based on its predicted *target address*. Some *hysteresis* bits are used to help guide the *taken/not-taken* decision of the BTB in the case of a *tag hit*. The BTB is a very expensive structure – for each BTB entry it must store the *tag* (anywhere from a partial tag of ≈20 bits to a full 64-bit tag[4]) and the *target* (a full 64 bit address[5]).

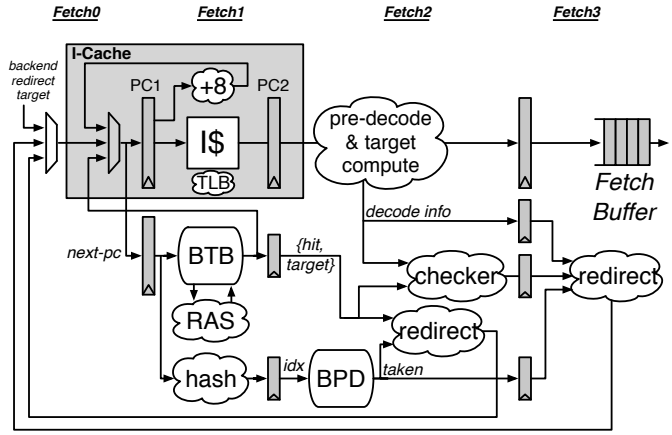
Return Address Stack (RAS) The RAS predicts function returns. *Jump-register* instructions are otherwise quite difficult to predict, as their target depends on a register value. However, functions are typically entered using a *Function Call* instruction at address A and return from the function using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

Conditional Branch Predictor (BPD). The BPD maintains a set of prediction and hysteresis tables to make *taken/not-taken* predictions based on a *look-up address*. The BPD *only* makes *taken/not-taken* predictions – it therefore relies on some other agent to provide information on what instructions are branches and what their targets are. The BPD can either use the BTB for this information or it can wait and decode the instructions themselves once they have been fetched from the instruction cache. Because the BPD does not store the expensive branch targets, it can be much denser and thus make more accurate predictions on the branch directions than the BTB – whereas each BTB entry may be 60 to 128

[4]Actually, less than 64 bits since few 64-bit machines will support using all 64-bits as part of the virtual address.
[5]An offset from the look-up address could be stored in the BTB table entry instead but that adds an *adder* to the critical path.
[6]Actually, it will be $A+4$ as the size of the call instruction to jump over is 4 bytes in RV64G.



(a) BOOMv1



(b) BOOMv2

Figure 3: The frontend pipelines for BOOMv1 and BOOMv2. To address critical path issues, BOOMv2 adds an extra stage. The branch predictor (BPD) index hashing function is moved to its own stage (F1), pushing back the BPD predictor table accesses a cycle as well (F2). BOOMv2 also utilizes a partially-tagged, set-associative BTB (BOOMv1 uses a fully-tagged fully-associative BTB). This requires adding a *checker* module which verifies that the predictions from the BTB matches the instructions being fetched.

bits, the BPD may be as few as one or two bits per branch.[7] A common arch-type of BPD is a *global history* predictor. Global history predictors work by tracking the outcome of the last N branches in the program ("global") and hashing this *history* with the *look-up address* to compute a look-up index into the BPD prediction tables. For a sophisticated BPD, this hashing function can become quite complex. BOOM's predictor tables are placed into single-ported SRAMs. Although many prediction tables are conceptually "tall and skinny" matrices (thousands of 2- or 4-bit entries), a generator written in Chisel transforms the predictor tables into a square memory structure to best match the SRAMs provided by a memory compiler.

Figure 3 shows the pipeline organization of the frontend. We found the a critical path in BOOMv1 to be the conditional branch predictor (BPD) making a prediction and redirecting the fetch instruction address in the F2 stage, as the BPD must first decode the newly fetched instructions and compute potential branch targets. For BOOMv2, we provide a full cycle to decode the instructions returning from the instruction cache and target computation (F2) and perform the redirection in the F3 stage. We also provide a full cycle for the hash indexing function, which removes the hashing off the critical path of *Next-PC selection*.

We have added the option for the BPD to continue to make predictions in F2 by using the BTB to provide the branch decode and target information. However, we found this path of accessing the prediction tables and redirecting the instruction stream in the same cycle requires a custom array design.

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

3.2 Distributed Issue Windows

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

The size of the issue window denotes the number of in-flight, un-executed instructions that can be selected for out-of-order execution. The larger the window, the more instructions the scheduler can attempt to re-order. For BOOM, the issue window is implemented as a collapsing queue to allow the oldest instructions to be compressed towards the top. For issue-select, a cascading priority encoder selects the oldest instruction that is ready to issue. This path is exacerbated either by increasing the number of entries to search across or by increasing the number of issue ports. For BOOMv1, our synthesizable implementation of a 20 entry issue window with three issue ports was found to be 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

[7] We are ignoring the fact that predictors such as TAGE [8] actually do store partial tags which can allow them to predict which instructions are branches.

path while also increasing the total number of instructions that can be scheduled. However, to maintain performance of executing two integer ALU instructions and one memory instruction per cycle, a common configuration of BOOM will use two issue-select ports on the integer issue window.

3.3 Register File Design

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

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

We took two different approaches to improving the register file. The first level was purely microarchitectural. We split apart *issue* and *register read* into two separate stages – *issue-select* is now given a full cycle to select and issue uops, and then another full cycle is given to read the operand data out of the register file. We lowered the register count by splitting up the unified physical register file into separate floating point and integer register files. This split also allowed us to reduce the read-port count by moving the three-operand fused-multiply add floating point unit to the smaller floating point register file.

The second path to improving the register file involved physical design. A significant problem in placing and routing a register file is the issue of *shorts* – a geometry violation in which two metal wires that should remain separate are physically attached. These shorts are caused by attempting to route too many wires to a relatively dense regfile array. BOOMv2's 70 entry integer register file of 6 read ports and 3 write ports comes to 4,480 bits, each needing 18 wires routed into and out of it. There is a mismatch between the synthesized array and the area needed to route all required wires, resulting in shorts.

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

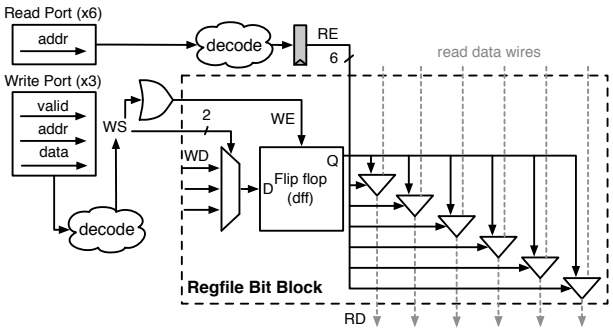


Figure 4: A Register File Bit manually crafted out of foundry-provided standard cells. Each read port provides a read-enable bit to signal a tri-state buffer to drive its port's read data line. The register file bits are laid out in an array for placement with guidance to the place tools. The tools are then allowed to automatically route the 18 wires into and out of each bit block.

ports inside of each cluster, and muxes select the read data across clusters.

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

4 LESSONS LEARNED

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

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

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

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

Describing hardware using generators proved to be a very useful technique; multiple design points could be generated and evaluated, and the final design choices could be committed to later in the design cycle. We could also increase our confidence that particular critical paths were worth pursuing; by removing functional units and register read ports, we

could estimate the improvement from microarchitectural techniques that would reduce port counts on the issue windows and register file.

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

5 WHAT DOES IT TAKE TO GO REALLY FAST?

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

For this analysis, we used regular threshold voltage (RVT)-based SRAM. However, the BTB is a crucial performance structure typically designed to be accessed and used to make a prediction within a single-cycle and is thus a candidate for additional custom effort. There are many other structures that are often the focus of manual attention: functional units; content-addressable memories (CAMs) are crucial for many structures in out-of-order processors such as the load/store unit or translation lookaside buffers (TLBs) [5]; and the issue-select logic can dictate how large of an issue window can be deployed and ultimately guide how many instructions can be inflight in an out-of-order processor.

However, any techniques to increase BOOM's clock frequency will have to be balanced against decreasing the IPC performance. For example, BOOM's new front-end suffers from additional bubbles even on correct branch predictions. Additional strategies will need to be employed to remove these bubbles when predictors are predicting correctly [9].

6 CONCLUSION

Modern out-of-order processors rely on a number of memory macros and arrays of different shapes and sizes, and many of them appear in the critical path. The impact on the 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 they may generate designs with poor performance and power characteristics. Re-architecting the design by relying on a hand-crafted, yet synthesizable register file array and leveraging hardware generators written in Chisel helped us isolate real critical paths from false ones. This methodology narrows down the range of arrays that would eventually have to be handcrafted for a serious production-quality implementation.

BOOM is still a work-in-progress and we can expect further refinements as we collect more data. As BOOMv2 has largely been an effort in improving the critical path of BOOM, there has been an expected drop in instruction per cycle (IPC) performance. Using the Coremark benchmark, we witnessed up to a 20% drop in IPC based on the parameters listed in Table 1. Over half of this performance degradation is due to the increased latency between load instructions and any dependent instructions. There are a number of available techniques to address this that BOOM does not currently employ. However, BOOMv2 adds a number of parameter options that allows the designer to configure the pipeline depth of the register renaming and register-read components, allowing BOOMv2 to recover most of the lost IPC in exchange for an increased clock period.

ACKNOWLEDGMENTS

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] Mark Anderson. 2010. A more cerebral cortex. *IEEE Spectrum* 47, 1 (2010).
[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225.
[3] Christopher Celio, Krste Asanović, and David A Patterson. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Technical Report. University of California, Berkeley.
[4] David G Chinnery and Kurt Keutzer. 2005. Closing the power gap between ASIC and custom: an ASIC perspective. In *Proceedings of the 42nd annual Design Automation Conference*. ACM, 275–280.
[5] Wei-Wu Hu, Ji-Ye Zhao, Shi-Qiang Zhong, Xu Yang, Elio Guidetti, and Chris Wu. 2007. Implementing a 1GHz four-issue out-of-order execution microprocessor in a standard cell ASIC methodology. *Journal of Computer Science and Technology* 22, 1 (2007), 1–14.
[6] R.E. Kessler. 1999. The Alpha 21264 Microprocessor. *IEEE Micro* 19, 2 (1999), 24–36.
[7] MIPS Technologies, Inc. 1996. *MIPS R10000 Microprocessor Users Manual*. Mountain View, CA.
[8] André Seznec. 2007. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* (2007).
[9] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. 1996. *Multiple-block ahead branch predictors*. Vol. 31. ACM.
[10] Steven JE Wilton and Norman P Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (1996), 677–688.
[11] K.C. Yeager. 1996. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro* 16, 2 (1996), 28–41.