The Berkeley Out-of-Order Machine (BOOM) Design Specification

Christopher Celio, David Patterson, and Krste Asanović
University of California, Berkeley, California 94720–1770
celio@eecs.berkeley.edu

January 11, 2016

This draft is a work-in-progress.

Contents

Chapter 1

Introduction & Overview

The goal of this document is to describe the design and implementation of the Berkeley Out–of–Order Machine (BOOM).

BOOM is heavily inspired by the MIPS R10k and the Alpha 21264 out–of–order processors[2, 3]. Like the R10k and the 21264, BOOM is a unified physical register file design (also known as "explicit register renaming").

The source code to BOOM can be found at (https://ucb-bar.github.io/riscv-boom).

1.1 The BOOM Pipeline

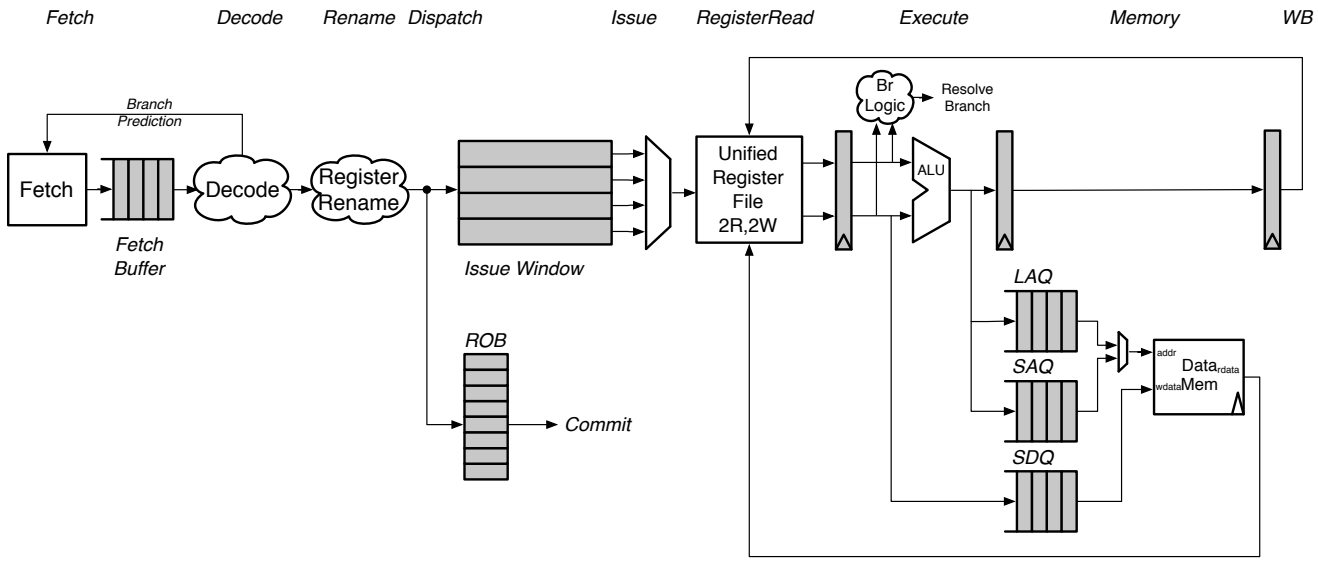

Figure 1.1: The Berkeley Out of Order Machine Processor.

Conceptually, BOOM is broken up into 10 stages: *Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback,* and *Commit.* However, many of those stages are combined in the current implementation, yielding *six* stages: *Fetch, Decode/Rename/Dispatch,*

Issue/RegisterRead, Execute, Memory, and *Writeback* (*Commit* occurs asynchronously, so I'm not counting that as part of the "pipeline").

Fetch Instructions are *fetched* from the Instruction Memory and placed into a four-entry deep FIFO, known as the *fetch buffer*.[1]

Decode *Decode* pulls instructions out of the *fetch buffer* and generates the appropriate "micro-op" to place into the pipeline.[2]

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

Dispatch The instruction is then *dispatched*, or written, into the *Issue Window*.

Issue Instructions sitting in the *Issue Window* wait until all of their operands are ready, and are then *issued*. This is the beginning of the out–of–order piece of the pipeline.

RF Read Issued instructions first *read* their operands from the unified physical register file...

Execute and then enter the *Execute* stage where the integer ALU resides. Issued memory operations perform their address calculations in the *Execute* stage, and then store the calculated addresses in the Load/Store Unit which resides in the *Memory* stage.

Memory The Load/Store Unit consists of three queues: a Load Address Queue (LAQ), a Store Address Queue (SAQ), and a Store Data Queue (SDQ). Loads are fired to memory when their address is present in the queue and does not conflict with any of the store addresses that the load depends on.[3] Stores are fired to memory at commit time, when both its address and its data are present.

Writeback ALU operations and load operations are *written* back to the physical register file.[4]

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

BOOM supports full branch speculation and branch prediction. Each instruction, no matter where it is in the pipeline, is accompanied by a branch tag that marks which branches the instruction is "speculated under". A mispredicted branch requires killing all instructions that depended on

[1] While the fetch buffer is four-entries deep, it can instantly read out the first instruction on the front of the FIFO. Put another way, instructions don't need to spend four cycles moving their way through the *fetch buffer* if there are no instructions in front of them.

[2] Because RISC-V is a RISC ISA, nearly all instructions generate only a single micro-op, with the exception of store instructions, which generate a "store address generation" micro-op and a "store data generation" micro-op.

[3] Technically, the load could bypass the data it needs out of the SDQ if it found a match in the SAQ. However, BOOM at this time does not support the bypassing of load data out of the SDQ. Problem **??** covers this issue in more detail.

[4] While BOOM is a single-issue processor, it does provide ALU operations and memory operations each their own write port, meaning the register file is a two-read, two-write register file (two different destinations can be written simultaneously).

that branch. When a branch instructions passes through *Rename*, copies of the *Register Rename Table* and the *Free List* are made. On a mispredict, the saved processor state is restored.

The *Decode* stage contains a Branch History Table, composed of simple n-bit history counters indexed by PC. On a predicted branch, the *Decode* stage kills the *fetch buffer* and redirects the *Fetch* stage. Otherwise, the *Fetch* stages fetches along PC+4.[5]

In this lab, BOOM implements a basic set of instructions from RV32. BOOM does not support sub-word memory accesses, floating point, or exceptions. Also, just like the simple pipelines from Lab 1, BOOM is connected to a magic, single-cycle memory (there are no caches or memory hierarchy). However, bypasses have been removed, meaning that the use-delay on back-to-back instructions is three cycles. Therefore, out–of–order issue is still an important component to good performance.

1.2 The RISC-V ISA

BOOM implements the RV64G variant of the RISC-V ISA. This includes the MAFD extensions and the privileged specification (multiply/divide, AMOs, load-reserve/store-conditional, single- and double-precision IEEE 754-2008 floating point). More information about the RISC-V ISA can be found at (`http://riscv.org`).

RISC-V provides the following features which make it easy to target with high-performance designs:

Relaxed memory model This greatly simplifies the Load/Store Unit, which does not need to have loads snoop other loads nor does coherence traffic need to snoop the LSU, as required by sequential consistency.

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

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

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

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

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

BOOM (currently) does not implement the "C" compressed extension nor the "V" vector extension.

[5] As of the writing of this handout, BOOM does *not* use a Branch Target Buffer, which would allow BOOM to redirect the PC in the *Fetch* stage.

1.3 The *Chisel* Hardware Construction Language

BOOM is implemented in the *Chisel* hardware construction language. More information about *Chisel* can be found at (http://chisel.eecs.berkeley.edu).

1.4 Quick-start

To build a BOOM C++ emulator and run BOOM through a couple of simple tests:

```
$ export ROCKETCHIP_ADDONS="boom"
$ git clone https://github.com/ucb-bar/rocket-chip.git
$ cd rocket-chip
$ git checkout boom
$ git submodule update --init
$ cd riscv-tools
$ git submodule update --init --recursive riscv-tests
$ cd ../emulator; make run CONFIG=BOOMCPPConfig
```

1.5 The BOOM Repository

The BOOM repository holds the source code to the BOOM core; it is not a full processor and thus is **NOT A SELF-RUNNING** repository. To instantiate a BOOM core, the Rocket chip generator found in the rocket-chip git repository must be used (https://github.com/ucb-bar/rocket-chip), which provides the caches, uncore, and other needed infrastructure to support a full processor.

The BOOM source code can be found in `boom/src/main/scala`.

The code structure is shown below:

- `boom/src/main/scala/`
  - bpd_pipeline.scala branch prediction stage.
  - brpredictor.scala abstract branch predictor.
  - configs.scala BOOM configurations.
  - consts.scala constant definitions.
  - core.scala the top-level of the processor core.
  - dcacheshim.scala the shim between the the core and the dcache.
  - decode.scala decode stage.
  - dpath.scala core datapath.
  - execute.scala high-level execution units (made up of FUs).
  - fpu.scala floating point unit.
  - functional_unit.scala low-level functional units.
  - gshare.scala gshare branch predictor.
  - imul.scala integer multiplier.
  - issue_ageordered.scala age-ordered (collasping-queue) issue window implementation.

- issue.scala abstract issue window.
- issue_slot.scala An issue window slot.
- issue_unordered.scala un-ordered issue window implementation.
- lsu.scala load/store unit.
- package.scala
- parameters.scala knobs/parameters.
- prefetcher.scala data prefetcher.
- regfile.scala register file.
- registerread.scala registerRead stage and bypassing.
- rename.scala register renaming logic.
- rob.scala re-order buffer
- tile.scala top-level tile.
- util.scala utility code.

1.6 The Rocket-chip Repository Layout

As BOOM is just a core, an entire SoC infrastructure must be provided. BOOM was developed to use the open-source Rocket-chip SoC generator (`https://github.com/ucb-bar/rocket-chip`). The Rocket-chip generator can instantiate a wide range of SoC designs, including cache-coherent multi-tile designs, cores with and without accelerators, and chips with or without a last-level shared cache.

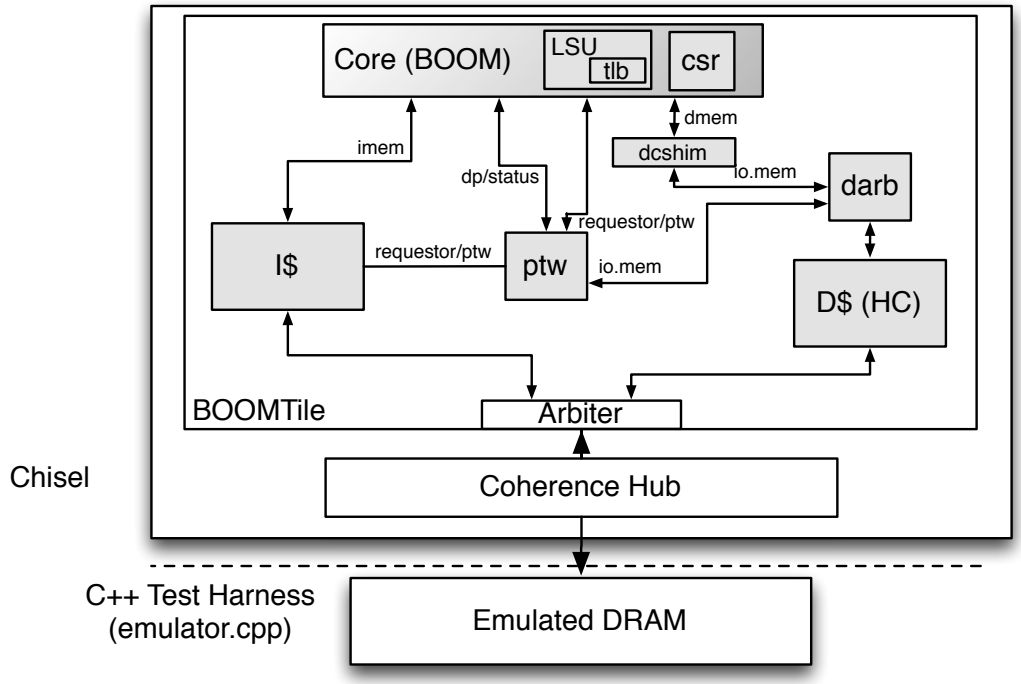

Figure 1.2: A single-core "BOOM-chip", with no L2 last-level cache.

To manage the wide array of actively developed projects that encompass Rocket-chip, the Rocket-chip repository makes heavy use of git submodules. The directory structure of the Rocket-chip repository is shown below.

- `${LAB3ROOT}`/
  - boom/ Git submodule of the *Chisel* source code for the BOOM core.
  - chisel The source code to the Chisel language itself.
  - emulator/ C++ simulation tools and support.
    - generated-src/Auto-generated C++ code.
    - Makefile Makefile for C++ simulation.
    - output/Output files from C++ simulation runs.
  - junctions/ Git submodule of the *Chisel* source code for the uncore and off-chip network.
  - riscv-tools/Git submodule that points to the RISC-V toolchain.
    - riscv-tests/ Source code for benchmarks and tests.
      - riscv-bmarks/ Benchmarks written in C.
      - riscv-tests/ Tests written in assembly.
  - Makefrag The high-level Makefile fragment.
  - src/ *Chisel* source code for top-level Rocket-chip.
  - rocket/ Git submodule of the *Chisel* source code for the Rocket core (used as a library of processor components).
  - sbt/ Chisel/Scala voodoo.
  - uncore/ Git submodule of the *Chisel* source code for the uncore components (including LLC).

Chapter 2

The Instruction Fetch Front-end

Chapter 3

Branch Prediction

This chapter discusses how BOOM predicts and resolves branch predictions.

BOOM uses two levels of branch prediction- a single-cycle "next-line predictor", and a slower but more complex "backing predictor".[1]

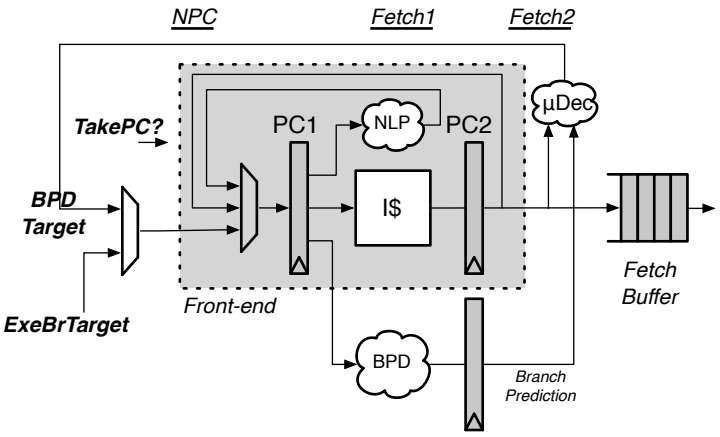

Figure 3.1: The Fetch Unit.

3.1 The Rocket Next-line Predictor (NLP)

BOOM instantiates the Rocket core's Front-End, which fetches instructions and predicts every cycle where to fetch the next instructions. If a misprediction is detected in BOOM's backend, or BOOM's own predictor wants to redirect the pipeline in a different direction, a request is sent to the Front-End and it begins fetching along a new instruction path.

The next-line predictor (NLP) takes in the current PC being used to fetch instructions (the *Fetch PC*) and predicts combinationally where the next instructions should be fetched for the next cycle. If predicted correctly, there are no pipeline bubbles.

[1] [TODO: Author: I'm open to better names!]

The next-line predictor is an amalgamation of a fully-associative branch target buffer (BTB), a gshare branch history table (BHT), and a return address stack (RAS).

3.1.1 NLP Predictions

The *Fetch PC* first performs a tag match to find a uniquely matching BTB entry. If a hit occurs, the BTB entry will make a prediction in concert with the BHT and RAS as to whether there is a branch, jump, or return found in the *fetch packet* and which instruction in the *fetch packet* is to blame. The BTB entry also contains a predicted PC target, which is used as the *Fetch PC* on the next cycle.

The hysteresis bits (governed by a gshare predictor) are only used on a BTB entry hit and if the predicting instruction is a branch.

If the BTB entry contains a *return* instruction, the RAS stack is used to provide the predicted return PC as the next *Fetch PC*. The actual RAS management is governed externally.

For area-efficiency, the high-order bits of the PC tags and PC targets are stored in a compressed file.

3.1.2 NLP Updates

Each branch passed down the pipeline remembers not only its own PC, but the *Fetch PC* used to fetch it.[2] Upon Branch Resolution, the NLP is updated with the outcome of the branch.

However, the branch's PC is not used for the BTB tag match, but rather the branch's *Fetch PC*. Even for superscalar fetching, only a single *Fetch PC* is passed to the NLP. The NLP must predict across the entire *fetch packet* which of the many possible branches will be the dominating branch that redirects the PC.

3.2 The Backing Predictor

[TODO: ...]

3.3 Branch Prediction Configurations

There are a number of parameters provided to govern the branch prediction in BOOM.

[TODO: ...]

[2] In reality, only the very lowest bits must be saved, as the higher-order bits will be the same.

Chapter 4

The Decode Stage

The decode stage takes instructions from the fetch buffer, decodes them, and allocates the necessary resources as required by each instruction. The decode stage will stall as needed if not all resources are available.

Chapter 5

The Rename Stage

The rename stage maps the *ISA* or *logical* register specifiers of each instruction to *physical* register specifiers.

Chapter 6

The Issue Unit

The issue window holds dispatched micro-ops that have not yet executed. When all of the operands for the micro-op are ready, the issue slot sets its "request" bit high. The issue select logic then chooses to issue a slot which is asserting its "request" signal. Once a micro-op is issued, it is removed from the issue window to make room for more dispatched instructions.

Future designs may choose to *speculatively* issue micro-ops (e.g., speculating that a load instruction will hit in the cache and thus issuing dependent micro-ops assuming the load data will be available in the bypass network). In such a scenario, the issue window cannot remove speculatively issued micro-ops until the speculation has been resolved. If a speculatively-issued micro-op failure occurs,

6.1 Issue Slot

Figure 6.1 shows a single issue slot from the *Issue Window*.[1]

Instructions (actually they are "micro-ops" by this stage) are *dispatched* into the *Issue Window*. From here, they wait for all of their operands to be ready ("p" stands for *presence* bit, which marks when an operand is *present* in the register file).

Once ready, the *issue slot* will assert its "request" signal, and wait to be *issued*. Currently, BOOM only issues a single micro-op every cycle, and has a fixed priority encoding to give the lower ID entries priority.

6.2 Issue Select Logic

6.3 Un-ordered Issue Window

6.4 Age-ordered Issue Window

[1] Conceptually, a bus is shown for implementing the driving of the signals sent to the *Register Read* Stage. In reality BOOM actually uses muxes.

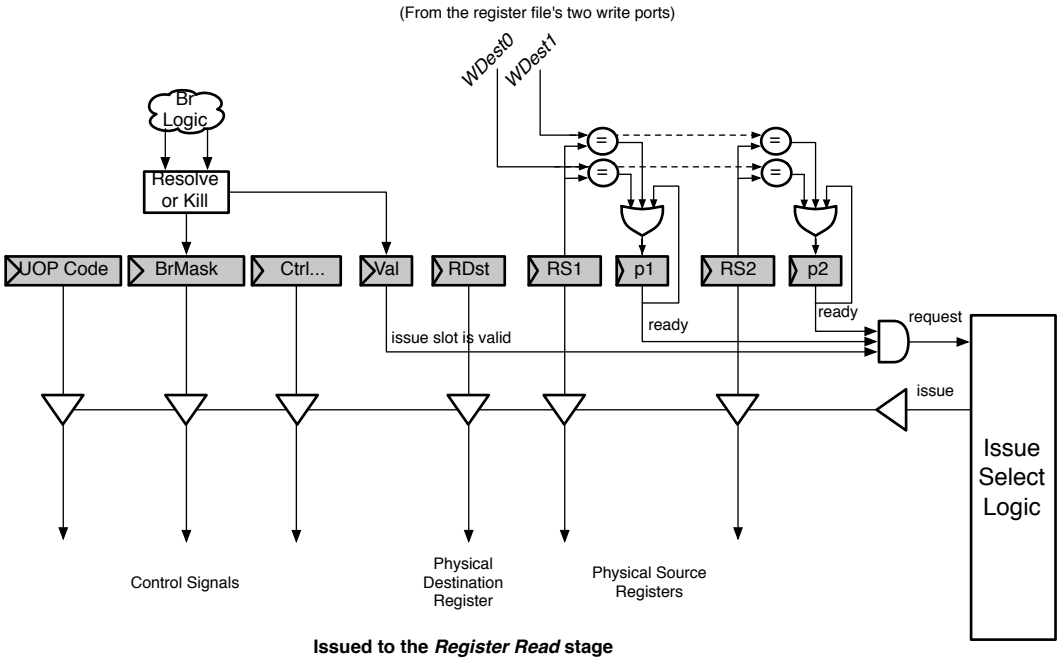

Figure 6.1: A single issue slot from the Issue Window.

Chapter 7

Register Read and Bypass

Chapter 8

The Execute Pipeline

Chapter 9

The Load/Store Unit (LSU)

The Load/Store Unit is responsible for deciding when to fire memory operations to the memory system. There are three queues: the Load Address Queue (LAQ), the Store Address Queue (SAQ), and the Store Data Queue (SDQ). Load instructions generate a "uopLD" micro-op. When issued, "uopLD" calculates the load address and places its result in the LAQ. Store instructions (may) generate *two* micro-ops, "uopSTA" (Store Address Generation) and "uopSTD" (Store Data Generation). The STA micro-op calculates the store address and places its result in the SAQ queue. The STD micro-op moves the store data from the register file to the SDQ. Each of these micro-ops will issue out of the *Issue Window* as soon their operands are ready. See Section 9.0.2 for more details on the store micro-op specifics.

9.0.1 Store Instructions

Entries in the Store Queue[1] are allocated in the *Decode* stage (the appropriate bit in the `stq_entry_val` vector is set). A "valid" bit denotes when an entry in the SAQ or SDQ holds a valid address or data (`saq_val` and `sdq_val` respectively). Once a store instruction is committed, the corresponding entry in the Store Queue is marked as committed. The store is then free to be fired to the memory system at its convenience. Stores are fired to the memory in program order.

9.0.2 Store Micro-ops

Stores are inserted into the issue window as a single instruction (as opposed to being broken up into separate `addr-gen` and `data-gen` micro-ops). This prevents wasteful usage of the expensive issue window entries and extra contention on the issue port to the LSU. A store in which both operands are ready can be issued to the LSU as a single micro-op which provides both the address and the data to the LSU. While this requires store instructions to have access to two register file read ports, this is motivated by a desire to not cut performance in half on store-heavy code. Sequences involving stores to the stack should operate at IPC=1!

However, it is common for store addresses to be known well in advance of the store data. Store addresses should be moved to the SAQ as soon as possible to allow later loads to avoid any memory ordering failures. Thus, the issue window will emit uopSTA or uopSTD micro-ops as required, but retain the remaining half of the store until the second operand is ready.

[1] When I refer to the *Store Queue*, I really mean both the SAQ and SDQ.

9.0.3 Load Instructions

Entries in the Load Queue (LAQ) are allocated in the *Decode* stage (`laq_entry_val`). In *Decode*, each load entry is also given a *store mask* (`laq_st_mask`), which marks which stores in the Store Queue the given load depends on. When a store is fired to memory and leaves the Store Queue, the appropriate bit in the *store mask* is cleared.

Once a load address has been computed and placed in the LAQ, the corresponding *valid* bit is set (`laq_val`).

Loads are optimistically fired to memory on arrival to the LSU (getting loads fired early is a huge benefit of out–of–order pipelines). Simultaneously, the load instruction compares its address with all of the store addresses that it depends on. If there is a match, the memory request is killed. If the corresponding store data is present, then the store data is *forwarded* to the load and the load marks itself as having *succeeded*. If the store data is not present, then the load goes to *sleep*. Loads that have been put to sleep are woken at *Commit* time.[2]

9.0.4 Memory Ordering Failures

The Load/Store Unit has to be careful regarding `s`tore→load dependences. For the best performance, loads need to be fired to memory as soon as possible.

`s`w x1 → 0(x2)

ld x3 ← 0(x4)

However, if `x`2 and `x`4 reference the same memory address, then the load in our example *depends* on the earlier store. If the load issues to memory before the store has been issued, the load will read the wrong value from memory, and a *memory ordering failure* has occurred. On an ordering failure, the pipeline must be flushed and the rename map tables reset. This is an incredibly expensive operation.

To discover ordering failures, when a store commits, it checks the entire LAQ for any address matches. If there is a match, the store checks to see if the load has *executed*, and if it got its data from memory or if the data was forwarded from an older store. In either case, a memory ordering failure has occurred.

See Figure A.1 for more information about the Load/Store Unit.

[2]Higher performance processors will track *why* a load was put to sleep and wake it up once the blocking cause has been alleviated.

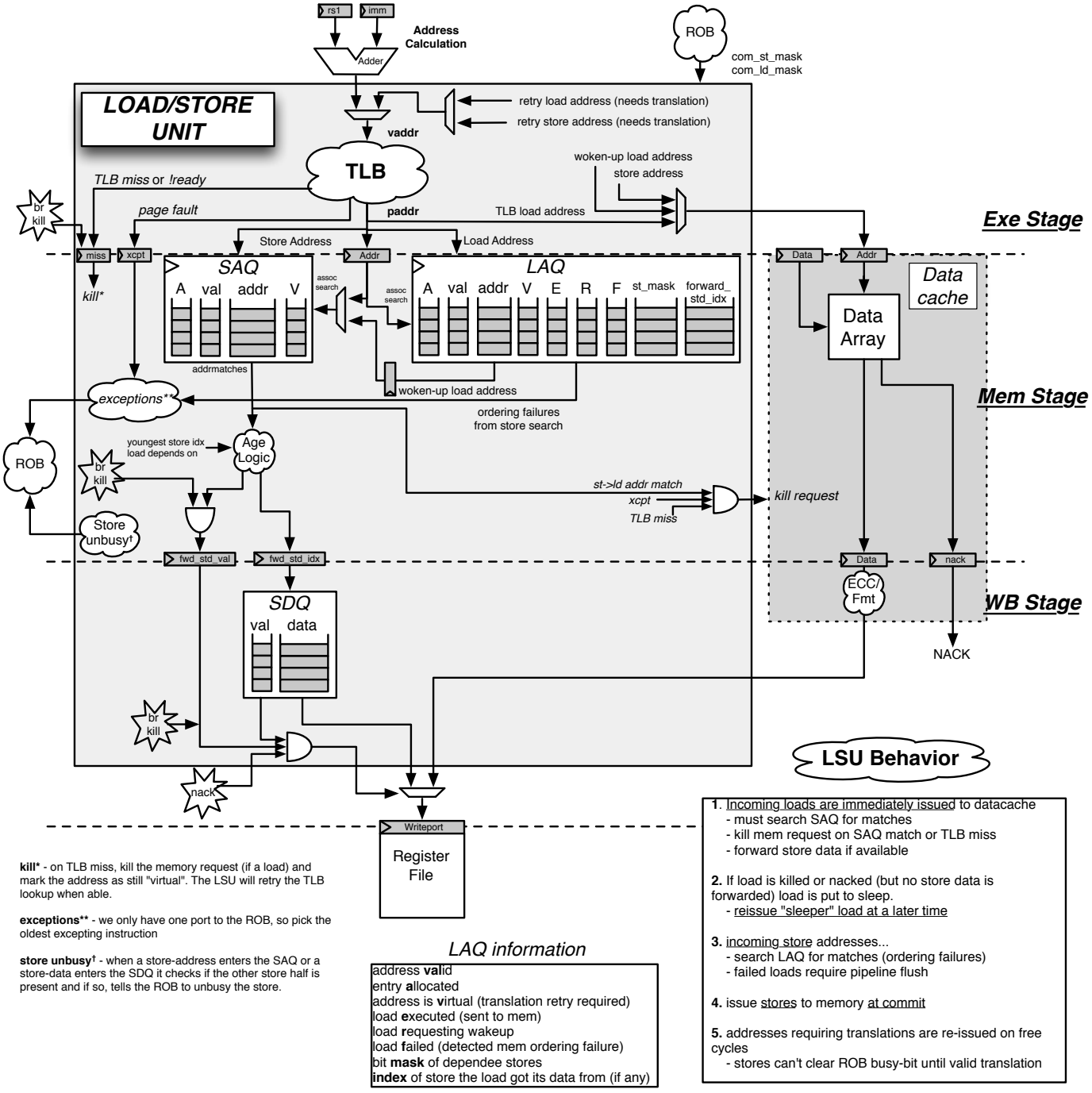

Figure 9.1: The Load/Store Unit.

Chapter 10

The Memory System and the Data-cache Shim

BOOM uses the Rocket non-blocking cache ("Hellacache"). Designed for use in in-order vector processors, a "shim" is used to connect BOOM to the data cache. The source code for the cache can be found in nbdcache.scala in the Rocket repository (github.com/ucb-bar/rocket).

The contract with the cache is that it may execute all memory operations sent to it (barring structural hazards). As BOOM will send speculative load instructions to the cache, the shim (dcacheshim.scala) must track all "inflight load requests" and their status. If an inflight load is discovered to be misspeculated, it is marked as such in the shim. Upon return from the data cache, the load's response to the pipeline is suppressed and it is removed from the inflight load queue.

The Hellacache does not ack store requests; the absence of a nack is used to signal a success.

All memory requests to the Hellacache may be killed the cycle after issuing the request (while the request is accessing the data arrays).

The current data cache design accesses the SRAMs in a single-cycle.

Chapter 11

The Reorder Buffer (ROB)

Chapter 12

Micro-architectural Counters

The current RISC-V gcc toolchain provides access to 16 named "uarch" counters.[1]

Table 12.1: Uarch Counters

Number	Event
0	Committed Branch Mispredictions
1	Committed Branches

The counters can be modified in `dpath.scala` to track events of interest.

Note: the counters can be quite large (64-bits each), and it is recommended that alternative methods be used if a silicon is the end-product. A design that can multiplex counters is recommended.

12.1 Reading UArch Counters in Software

The Code Example 12.1 demonstrates how to read the value of any CSR register from software.

```
1 #define read_csr_safe(reg) ({ register long __tmp asm("a0"); \
2   asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
3   __tmp; })
4
5   long csr_cycle   = read_csr_safe(cycle);
6   long csr_instr   = read_csr_safe(instret);
7   long csr_uarch0  = read_csr_safe(uarch0);
8   ...
9   long csr_uarch15 = read_csr_safe(uarch15);
```

Code 12.1: Reading a CSR register

[1]The future plan of record is to move these counters into the memory-mapped access region. This will open up the ability to track a much larger number of counters, as desired.

Chapter 13

Verification

This chapter covers the current recommended techniques for verifying BOOM. Although not provided as part of the BOOM or rocket-chip repositories, it is also recommended that BOOM be tested on "hello-world + riscv-pk" and the RISC-V port of Linux to properly stress the processor.

13.1 RISC-V Tests

A basic set of functional tests and micro-benchmarks can be found at (https://github.com/riscv/riscv-tests). These are invoked by the "make run" targets in the emulator, fsim, and vsim directories.

13.2 RISC-V Torture Tester

Berkeley's riscv-torture tool is used to stress the BOOM pipeline, find bugs, and provide small code snippets that can be used to debug the processor. Torture can be found at (https://github.com/ucb-bar/riscv-torture).

Quick-start

```
# compile the BOOM C++ emulator
$ cd rocket-chip/emulator; make run CONFIG=BOOMCPPConfig


# check out and run the riscv-torture repository
$ cd ../          # top-level rocket-chip directory
$ git clone https://github.com/ucb-bar/riscv-torture.git
$ cd riscv-torture
$ git submodule update --init
$ vim Makefile    # change RTL_CONFIG=BOOMCPPConfig
$ make igentest   # test that torture works, gen a single test
$ make cnight     # run C++ emulator overnight
```

Chapter 14

Debugging

Appendix A

Future Work

This chapter lays out some of the potential future directions that BOOM can be taken. To help facilitate such work, the preliminary design sketches are described below.

A.1 The ROcket Custom Co-processor Interface (ROCC)

The Rocket in-order processor comes with a ROCC interface that facilitates communication with co-processor/accelerators. Such accelerators include crypto units (e.g., SHA3) and vector processing units (e.g., the open-source Hwacha vector-thread unit[1]).

The ROCC interface accepts co-processor commands emitted by *committed* instructions run on the "Control Processor" (e.g., a scalar Rocket core). Any ROCC commands *will* be executed by the co-processor (barring exceptions thrown by the co-processor); nothing speculative can be issued over ROCC.

Some ROCC instructions will write back data to the Control Processor's scalar register file.

A.1.1 The Demands of the ROCC Interface

The ROCC interface accepts a ROCC command and up to two register inputs from the Control Processor's scalar register file. The ROCC command is actually the entire RISC-V instruction fetched by the Control Processor (a "ROCC instruction"). Thus, each ROCC queue entry is at least 2*XPRLEN + 32 bits in size (additional ROCC instructions may use the longer instruction formats to encode additional behaviors). [TODO: draw a diagram showing this]

As BOOM does not store the instruction bits in the ROB, a separate data structure (A "ROCC Reservation Station") will have to hold the instructions until the ROCC instruction can be committed and the ROCC command sent to the co-processor.

The source operands will also require access to BOOM's register file. Two possibilities are proposed:

- ROCC instructions are dispatched to the Issue Window, and scheduled so that they may access the read ports of the register file once the operands are available. The operands are then written into the ROCC Reservation Station, which stores the operands and the instruction bits until they can be sent to the co-processor. This may require significant state.

- ROCC instructions, when they are committed and sent to the ROCC command queue, must somehow access the register file to read out its operands. If the register file has dynamically scheduled read ports, this may be trivial. Otherwise, some technique to either inject a ROCC micro-op into the issue window or a way to stall the issue window while ROCC accesses the register file will be needed.

A.1.2 A Simple One-at-a-Time ROCC Implementation

The simplest way to add ROCC support to BOOM would be to stall *Decode* on every ROCC instruction and wait for the ROB to empty. Once the ROB is empty, the ROCC instruction can proceed down the BOOM pipeline non-speculatively, and get sent to the ROCC command queue. BOOM remains stalled until the ROCC accelerator acknowledges the completion of the ROCC instruction and sends back any data to BOOM's register file. Only then can BOOM proceed with its own instructions.

A.1.3 A High-performance ROCC Implementation Using Two-Phase Commit

While some of the above constraints can be relaxed, the performance of a decoupled co-processor depends on being able to queue up multiple commands while the Control Processor runs ahead (prefetching data and queueing up as many commands as possible). However, this requirement runs counter to the idea of only sending committed ROCC instructions to the co-processor.

BOOM's ROB can be augmented to track *commit* and *non-speculative* pointers. The *commit* head pointer tracks the next instruction that BOOM will *commit*, i.e., the instruction that will be removed from the ROB and the resources allocated for that instruction will be de-allocated for use by incoming instructions. The *non-speculative* head will track which instructions can no longer throw an exception and are no longer speculated under a branch (or other speculative event), i.e., which instructions absolutely will execute and will not throw a pipeline-retry exception.

This augmentation will allow ROCC instructions to be sent to the ROCC command queue once they are deemed "non-speculative", but the resources they allocate will not be freed until the ROCC instruction returns an acknowledgement. This prevents a ROCC instruction that writes a scalar register in BOOM's register file from overwriting a newer instruction's writeback value, a scenario that can occur if the ROCC instruction commits too early, followed by another instruction committing that uses the same ISA register as its writeback destination.

A.1.4 The BOOM Custom Co-processor Interface (BOCC)

Some accelerators may wish to take advantage of speculative instructions (or even out-of-order issue) to begin executing instructions earlier to maximize de-coupling. Speculation can be handled by either by epoch tags (if in-order issue is maintained to the co-processor) or by allocating mask bits (to allow for fine-grain killing of instructions).

A.2 The Vector ("V") ISA Extension

Implementing the Vector Extension in BOOM would open up the ability to leverage performance (or energy-efficiency) improvements in running data-level parallel codes (DLP).

While it would be relatively easy to add vector arithmetic operations to BOOM, the significant challenges lay in the vector load/store unit.

[TODO: ...]

A.3 The Compressed ("C") ISA Extension

This section describes how to approach adding the Compressed ISA Extension to BOOM. The Compressed ISA Extension, or RVC (http://riscv.org/download.html#spec_compressed_isa) enables smaller, 16 bit encodings of common instructions to decrease the static and dynamic code size. "RVC" comes with a number of features that are of particular interest to micro-architects:

- All 16b instructions map directly into a longer 32b instruction.
- 32b instructions have no alignment requirement, and may start on a half-word boundary.

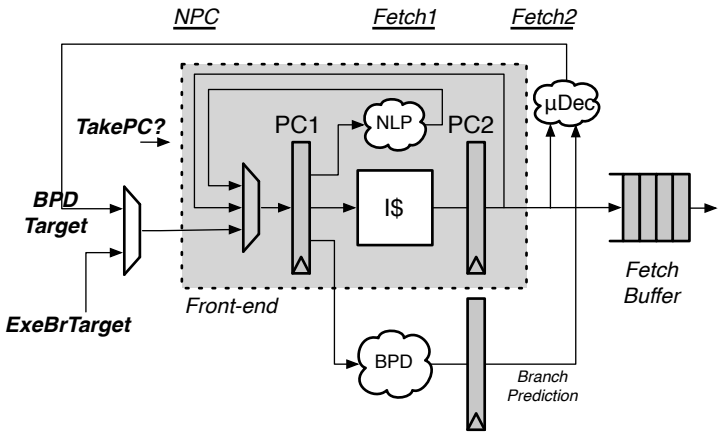

Figure A.1: The Fetch Unit. The grey box encompasses the Rocket front-end, which is re-used by BOOM.

BOOM re-uses the front-end design from Rocket, a 5-stage in-order core. BOOM then takes instructions returning (the *fetch packet*) from the Rocket front-end, quickly decodes the instructions for branch prediction, and pushes the *fetch packet* into the *Fetch Buffer*.

The C Extension provides the following challenges to micro-architects:

- Increased decoding complexity (e.g., operands can now move around).
- Finding *where* the instruction begins.
- Tracking down +4 assumptions throughout the code base, particularly with branch handling.
- Unaligned instructions, in particular, running off cache lines and virtual pages.

The last point requires some additional "statefulness" in the Fetch Unit, as fetching all of the pieces of an instruction may take multiple cycles.

The following describes the proposed implementation of RVC for BOOM:

- Implement RVC in the Rocket in-order core. Done properly, BOOM will then gain RVC support almost entirely for free (modulo any +4 assumptions in the code base).
- Move BOOM's *Fetch Buffer* into Rocket's front-end. Rocket will need the statefulness to handle wrap-around issues with fetching unaligned 32 bit instructions. A non-RVC Rocket core can optionally remove this buffer.
- Expand 16-bit instructions as they enter (or possibly exit) the *Fetch Buffer*.
- Minimize latency by placing 16b→32b expanders at every half-word start.

Appendix B

Frequently Asked Questions

To be filled in as questions are asked!

Appendix C

Terminology

To be filled in as needed.

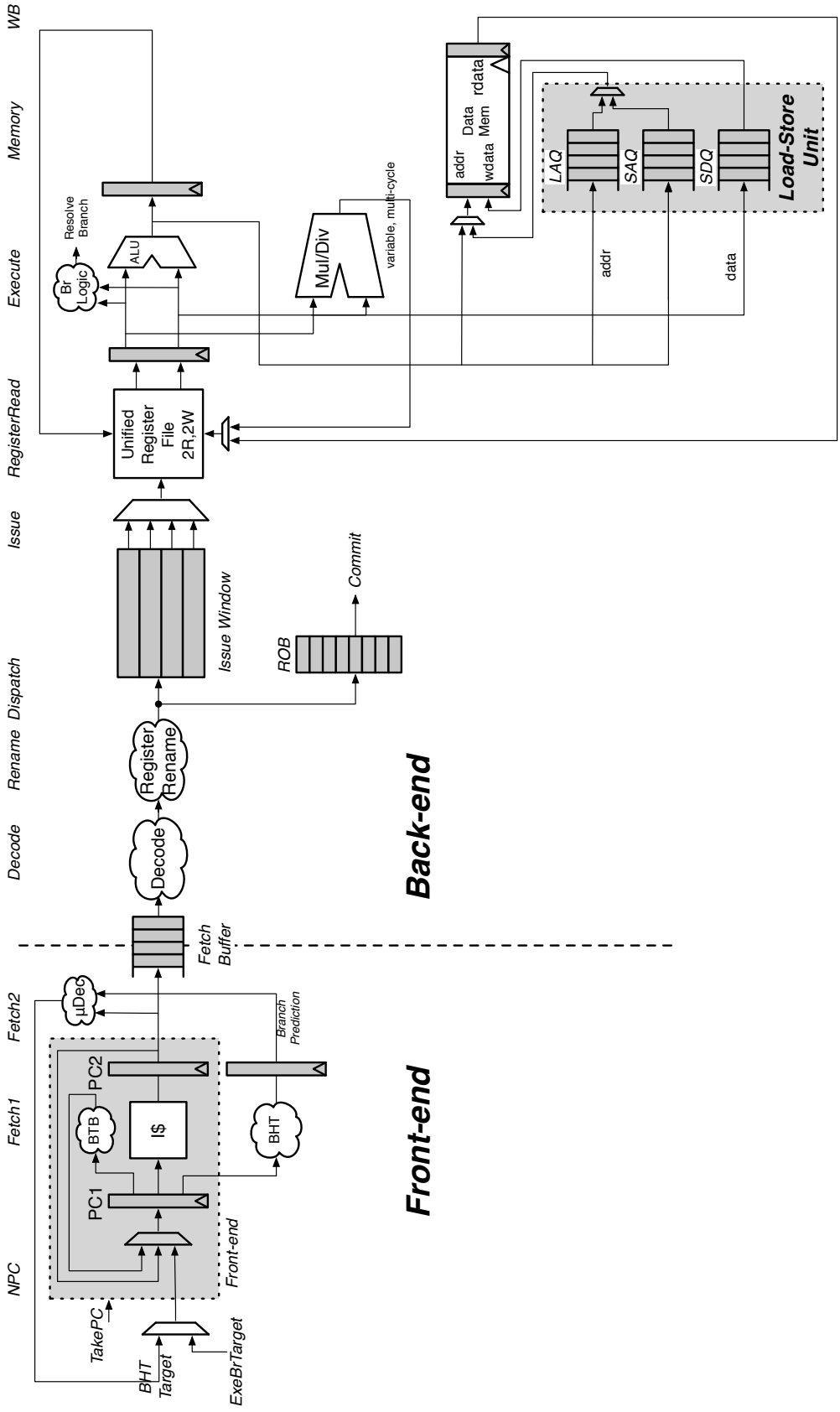

Figure C.1: A more detailed diagram of BOOM, for a single-issue RV32IM design.

Bibliography

[1] The Hwacha Project, 2015. http://hwacha.org.
[2] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, 1999.
[3] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, 1996.