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 20, 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

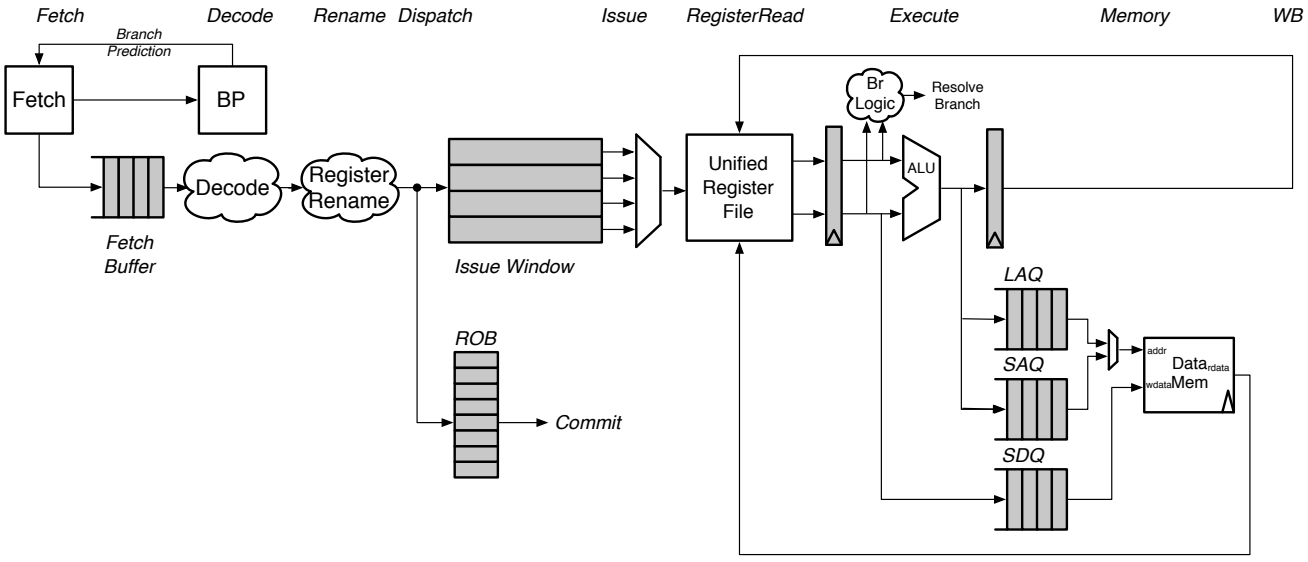

Figure 1.1: The Berkeley Out of Order Machine Processor.

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 FIFO queue, 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*.[3] 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 (or from the bypass network)...

Execute and then enter the *Execute* stage where the functional units reside. 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. 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.

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 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.

Although Figure 1.1 shows a simplified pipeline, BOOM implements the RV64G and privileged ISAs, which includes single- and double-precision floating point, atomic memory support, and page-based virtual memory.

[1] While the fetch buffer is N-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 N 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, currently all instructions generate only a single micro-op. More details on how store micro-ops are handled can be found in Chapter 11.

[3] More precisely, uops that are ready assert their request, and the issue scheduler chooses which uops to issue that cycle.

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 of 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.

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
```

Note: this assumes you have already installed the riscv-tools toolchain. If not, visit (https://github.com/riscv/riscv-tools).

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.

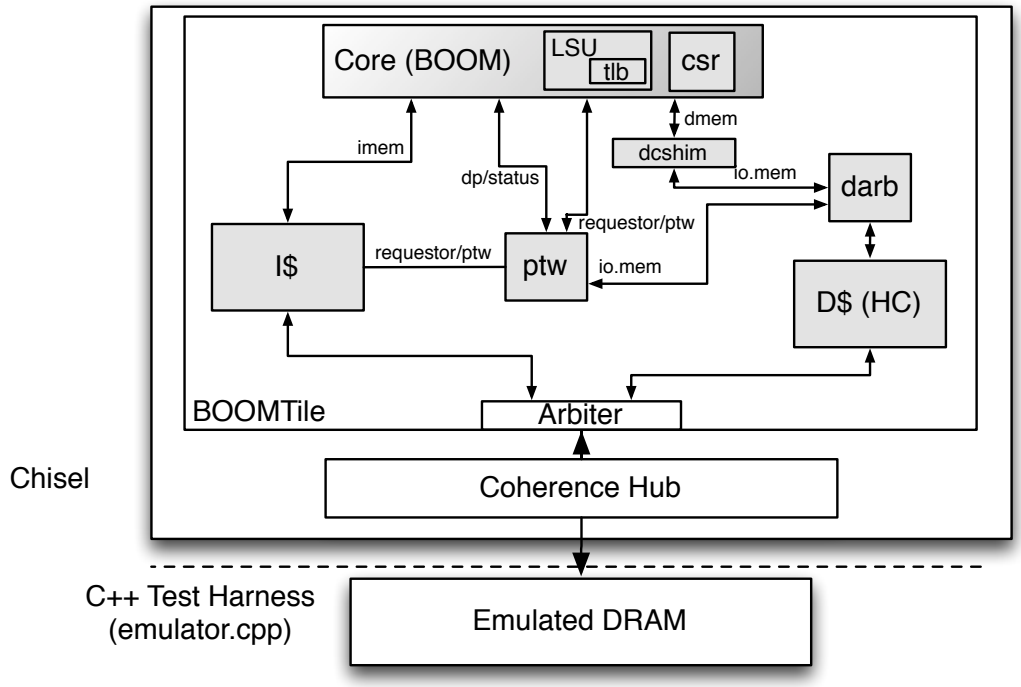

Figure 1.2: A single-core "BOOM-chip", with no L2 last-level cache.

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.

- `rocket-chip/`
  - boom/ Git submodule of the *Chisel* source code for the BOOM core.
  - chisel The source code to the Chisel language itself.
  - csrc/ Utility C/C++ source code.
  - dramsim2/ Git submodule of a C++ DRAM emulator.
  - emulator/ C++ simulation tools and support.
    - generated-src/Auto-generated C++ code.
    - Makefile Makefile for C++ simulation.
    - output/Output files from C++ simulation runs.
  - fpga-zynq/ Git submodule of the zynq FPGA tools.
  - fsim/ The FPGA Verilog simulation and build directories.
  - 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).
  - vsim/ The ASIC Verilog simulation and build directories.
  - zscale/ Git submodule of the *Chisel* source code for the zscale micro-controller core.

1.6.1 The Rocket Core - a Library of Processor Components!

Rocket is a 5-stage in-order core that implements the RV64G ISA and page-based virtual memory. The original design purpose of the Rocket core was to enable architectural research into vector co-processors by serving as the scalar *Control Processor*. Some of that work can be found at (`http://hwacha.org`).

Rocket has been taped out at least ten times in two different commercial processes, and has been successfully demonstrated to reach over 1.65 GHz in IBM 45 nm SOI.[**?**] As its namesake suggests, Rocket is the baseline core for the Rocket-chip SoC generator. As discussed earlier, BOOM is instantiated by replacing a Rocket tile with a BOOM tile.

However, from BOOM's point of view, Rocket can also be thought of as a "Library of Processor Components." There are a number of modules created for Rocket that are also used by BOOM - the functional units, the caches, the translation look-aside buffers, the page table walker, and more. Thus, throughout this document you will find references to these Rocket components and descriptions on how they fit into BOOM.

The source code to Rocket can be found at (`https://github.com/ucb-bar/rocket`).

Chapter 2

Instruction Fetch

Figure 3.1 shows the Fetch Unit organization used by BOOM.

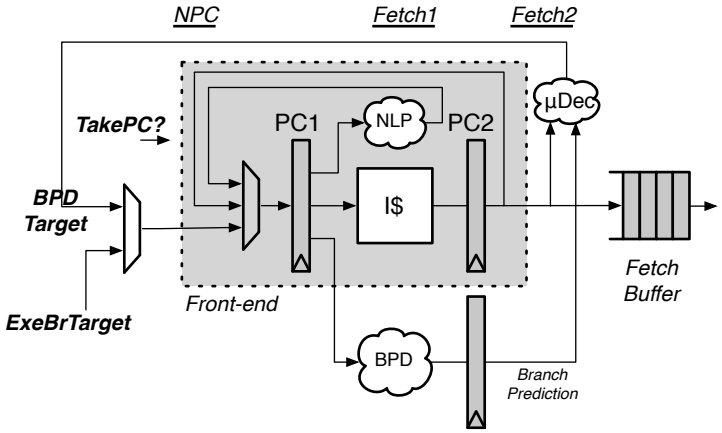

Figure 2.1: The Fetch Unit. The grey box is the front-end instantiated from the Rocket code base.

BOOM instantiates the Rocket core's *Front-end* (highlighted in grey in Fig 3.1), which fetches instructions and predicts every cycle where to fetch the next instructions using a "next-line predictor" (NLP). 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. See Chapter 3 for more information on how branch prediction fits into the Fetch Unit's pipeline.

As superscalar fetch is supported, the *Front-end* returns a *fetch packet* of instructions. The *fetch packet* also contains meta-data, which includes a *valid mask* (which instructions in the packet are valid?) and some branch prediction information that is used later in the pipeline.

2.1 The Rocket I-Cache

BOOM instantiates the i-cache found in the Rocket processor source code. The i-cache is a virtually indexed, physically tagged set-associative cache.

To save power, the i-cache reads out a fixed number of bytes (aligned) and stores the instruction bits into a register. Further instruction fetches can be managed by this register. The i-cache is only fired up again once the fetch register has been exhausted (or a branch prediction directs the PC elsewhere).

The i-cache does not (currently) support fetching across cache-lines, nor does it support fetching unaligned relative to the superscalar fetch address.[1]

The i-cache does not (currently) support hit-under-miss. If an icache miss occurs, the icache will not accept any further requests until the miss has been handled. This is less than ideal for scenarios in which the pipeline discovers a branch mispredict and would like to redirect the icache to start fetching along the correct path.

The front-end (currently) only handles the RV64G ISA, which uses fixed-size 4 bytes instructions.

2.2 The Fetch Buffer

Fetch packets coming from the i-cache are placed into a *Fetch Buffer*. The *Fetch Buffer* helps to decouple the instruction fetch front-end from the execution pipeline in the back-end.

The instructions within a *fetch packet* are *not* collapsed or compressed - any bubbles within a *fetch packet* are maintained.

The *Fetch Buffer* is parameterizable. The number of entries can be changed and whether the buffer is implemented as a "flow-through" queue[2] or not can be toggled.

[1] This constraint is due to the fact that a cache-line is not stored in a single row of the memory bank, but rather is striped across a single bank to match the refill size coming from the uncore. Fetching unaligned would require modification of the underlying implementation, such as banking the i-cache such that consecutive chunks of a cache-line could be accessed simultaneously.

[2] A flow-through queue allows entries being enqueued to be immediately dequeued if the queue is empty and the consumer is requesting (the packet "flows through" instantly).

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]

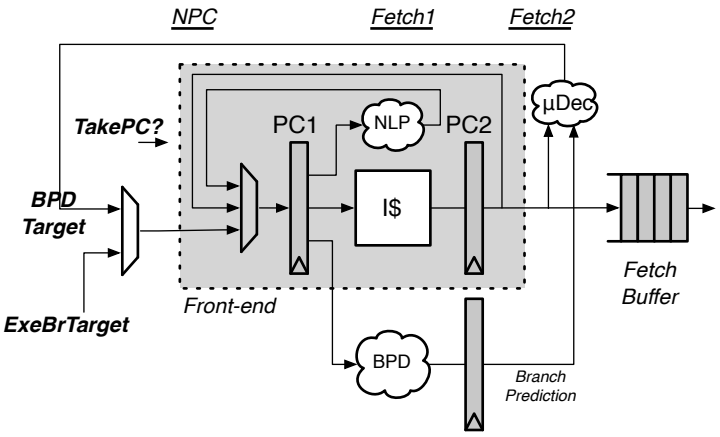

Figure 3.1: The Fetch Unit.

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.

5.1 The Purpose of Renaming

Renaming is a technique to rename the ISA (or *logical*) register specifiers in an instruction by mapping them to a new space of *physical* registers. The goal to *register renaming* is to break the output- (WAW) and anti-dependences (WAR) between instructions, leaving only the true dependences. Said again, but in architectural terminology, register renaming eliminates write-after-write (WAW) and write-after-read (WAR) hazards, which are artifacts introduced by a) only having a limited number of ISA registers to use as specifiers and b) loops, which by their very nature will use the same register specifiers on every loop iteration.

5.2 The Explicit Renaming Design

BOOM is an "explicit renaming" or "physical register file" out-of-order core design. A physical register file, containing many more registers than the ISA dictates, holds both the committed architectural register state and speculative register state. The rename map tables contain the information needed to recover the committed state. As instructions are renamed, their register specifiers are explicitly updated to point to physical registers located in the physical register file. The MIPS R10k, Alpha 21264, Intel Sandy Bridge, and ARM Cortex A15 cores are all example of explicit renaming out-of-order cores.

This is in contrast to an "implicit renaming" or "data-in-ROB" out-of-order core design. The Architectural Register File (ARF) only holds the committed register state, while the ROB holds the speculative write-back data. On commit, the ROB transfers the speculative data to the ARF.

The Pentium 4 and the ARM Cortex A57 are examples of *implicit renaming* designs.

5.3 The Rename Map Tables

The Rename Map Tables hold the speculative mappings from ISA registers to physical registers.

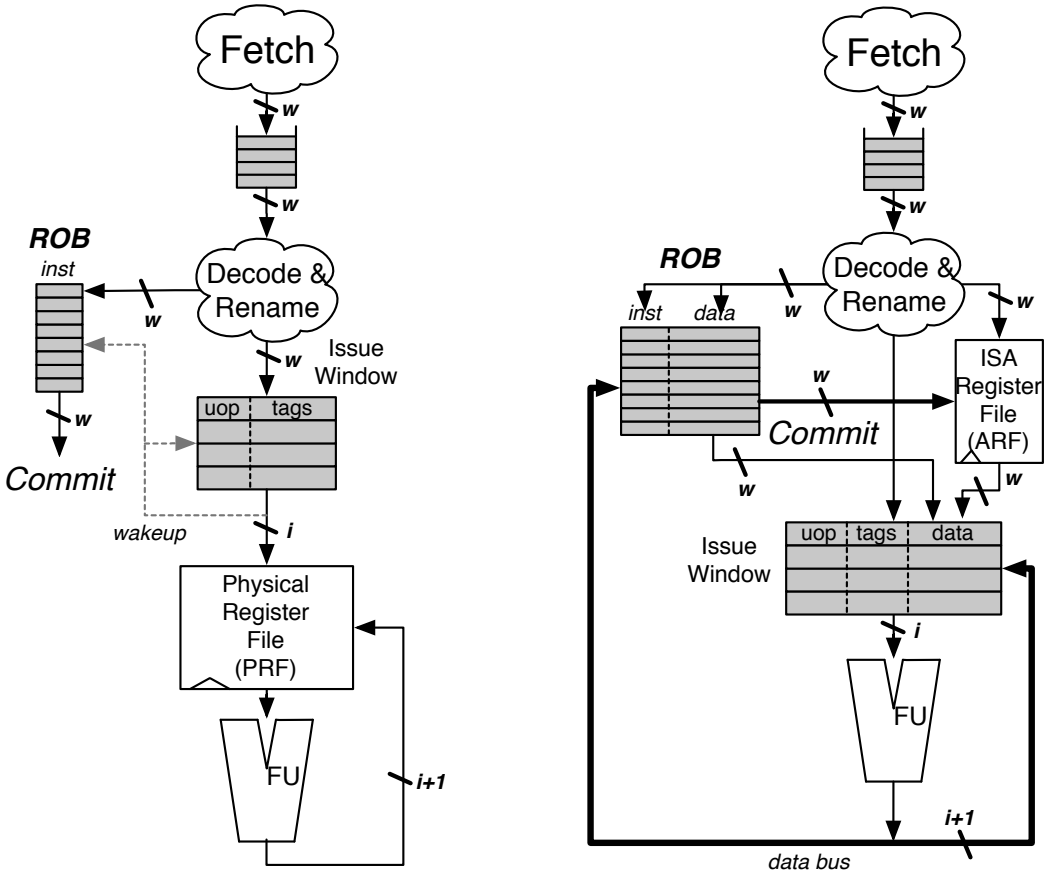

Figure 5.1: A PRF design (left) and a data-in-ROB design (right).

Each branch gets its own copy of the rename map tables.[1] On a branch mispredict, the map table can be reset instantly from the mispredicting branch's copy of the map table.

As the RV64G ISA uses fixed locations of the register specifiers (and no implicit register specifiers), the map tables can be read before the instruction is decoded!

5.3.1 Resets on Exceptions and Flushes

An additional, optional "Committed Map Table" holds the rename map for the committed architectural state. If enabled, this allows single-cycle reset of the pipeline during flushes and exceptions (the current map table is reset to the committed map table).

5.4 The Busy Table

The Busy Table tracks the readiness status of each physical register. If all physical operands are ready, the instruction will be ready to be issued.

[1] An alternate design for wider pipelines may prefer to only make up to one snapshot per cycle, but this comes with additional complexity to deduce the precise mappings for any given instruction within the fetch packet.

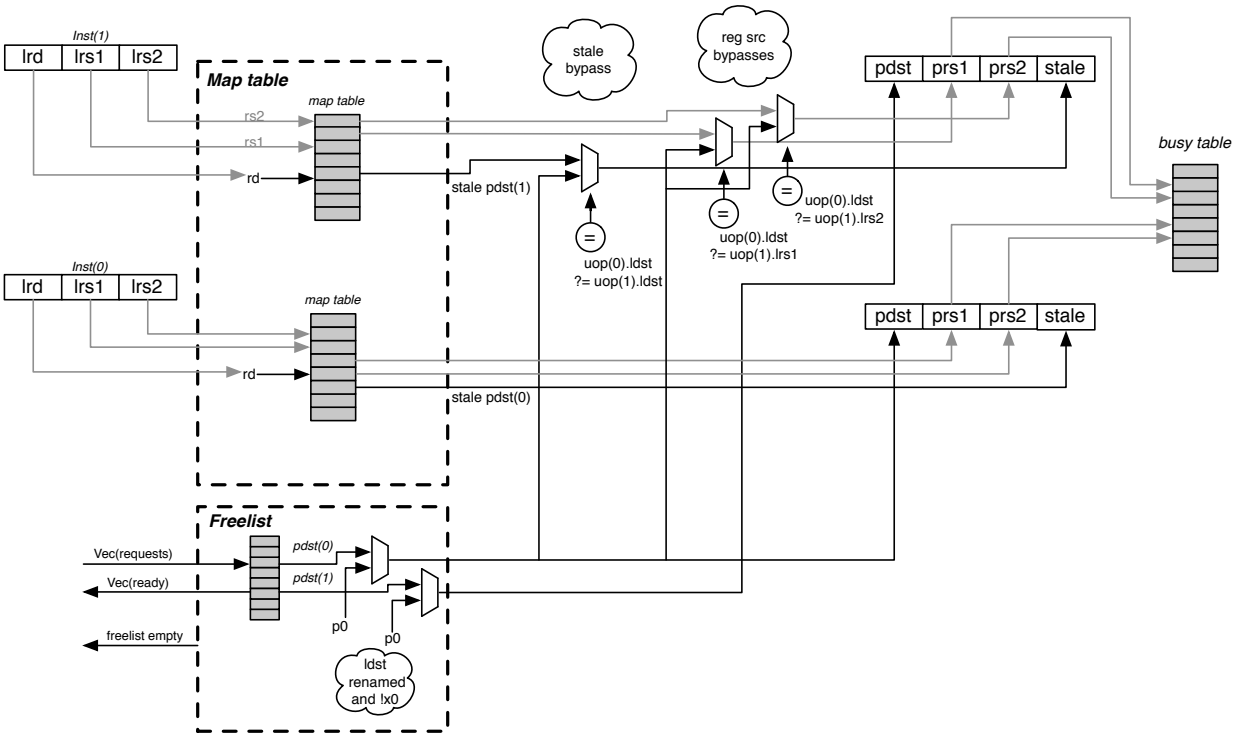

Figure 5.2: The Rename Stage. Logical register specifiers read the map table to get their physical specifier. For superscalar rename, any changes to the map tables must be bypassed to dependent instructions. The physical source specifiers can then read the Busy Table. The *Stale* specifier is used to track which physical register will be freed when the instruction later commits. P0 in the Physical Register File is always 0.

5.5 The Free List

The free-list tracks the physical registers that are currently un-used and is used to allocate new physical registers to instructions passing through the *Rename* stage.

The Free List is implemented as a bit-vector. A priority decoder can then be used to find the first free register. BOOM uses a cascading priority decoder to allocate multiple registers per cycle.[2]

On every branch (or jalr), the rename map tables are snapshotted to allow single-cycle recovery on a branch misprediction. Likewise, the Free List also sets aside a new "Allocation List", initialized to zero. As new physical registers are allocated, the Allocation List for each branch is updated to track all of the physical registers that have been allocated after the branch. If a misspeculation occurs, its Allocation List is added back to the Free List by *OR'ing* the branch's Allocation List with the Free List.

5.6 Stale Destination Specifiers

For instructions that will write a register, the map table is read to get the *stale physical destination specifier* ("stale pdst"). Once the instruction commits, the *stale pdst* is returned to the free list, as no future instructions will read it.

[2] A two-wide rename stage could use two priority decoders starting from opposite ends.

Chapter 6

The Reorder Buffer (ROB) and the Dispatch Stage

The ROB tracks the state of all inflight instructions in the pipeline. The role of the ROB is to provide the illusion to the programmer that his program executes in-order. After instructions are decode and renamed, they are then dispatched to the ROB and the issue window and marked as *busy*. As instructions finish execution, they inform the ROB and are marked *not busy*. Once the "head" of the ROB is no longer busy, the instruction is *committed*, and it's architectural state now visible. If an exception occurs and the excepting instruction is at the head of the ROB, the pipeline is flushed and no architectural changes that occurred after the excepting instruction are not made visible. The ROB then redirects the PC to the appropriate exception handler.

6.1 The ROB Organization

The ROB is, conceptually, a circular buffer that tracks all inflight instructions in-order. The oldest instruction is pointed to by the *commit head*, and the newest instruction will be added at the *rob tail*.

To facilitate superscalar *Dispatch* and *Commit*, the ROB is implemented as a circular buffer with W banks (where W is the *dispatch* and *commit* width of the machine). This organization is shown in Figure 6.1.

At *dispatch*, up to W instructions are written from the *fetch packet* into an ROB row, where each instruction is written to a different bank across the row. As the instructions within a *fetch packet* are all consecutive (and aligned) in memory, this allows a single PC to be associated with the entire *fetch packet* (and the instruction's position within the *fetch packet* provides the low-order bits to its own PC). While this means that branching code will leave

6.2 ROB State

Each ROB entry contains relatively little state:

- is entry valid?
- is entry busy?

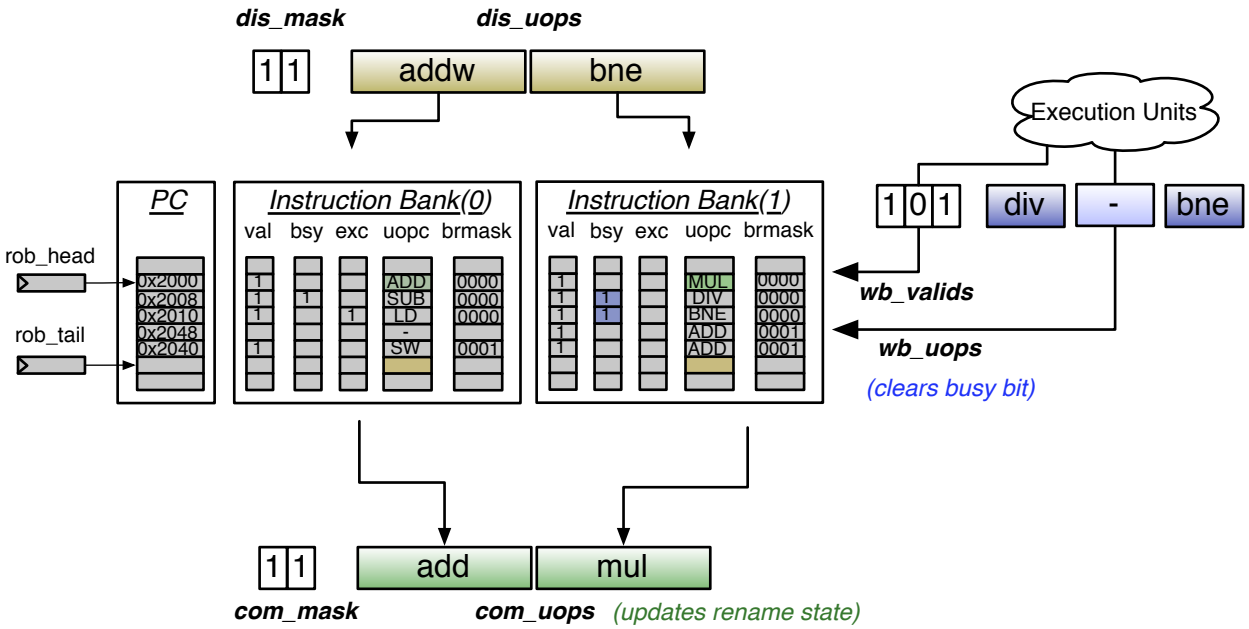

Figure 6.1: The Reorder Buffer for a two-wide BOOM with three-issue. Dispatched uops (*dis_uops*) are written at the bottom of the ROB (*rob_tail*), while committed uops (*com_uops*) are committed from the top, at *rob_head*, and update the rename state. Uops that finish executing (*wb_uops*) clear their *busy* bit. **Note:** the dispatched uops are written into the same ROB row together, and are located consecutively in memory allowing a single PC to represent the entire row.

- is entry an exception?
- branch mask (which branches is this entry still speculated under?
- rename state (what is the logical destination and the stale physical destination?)
- floating-point status updates
- other miscellaneous data (e.g., helpful for statistic tracking)

The PC and the branch prediction information is stored on a per-row basis (see Sec 6.2.2). The Exception State only tracks the oldest known excepting instruction (see Sec 6.2.1).

6.2.1 Exception State

The ROB tracks the oldest excepting instruction. If this instruction reaches the head of the ROB, then an exception is thrown.

Each ROB entry is marked with a single-bit to signify whether or not the instruction has encountered exceptional behavior, but the additional exception state (e.g., the bad virtual address and the exception cause) is only tracked for the oldest known excepting instruction. This saves considerable state by not storing this on a per entry basis.

6.2.2 PC Storage

The ROB must know the PC of every inflight instruction. This information is used in the following situations:

- Any instruction could cause an exception, in which the "exception pc" (epc) must be known.
- Branch and jump instructions need to know their own PC for for target calculation.
- Jump-register instructions must know both their own PC **and the PC of the following instruction** in the program to verify if the front-end predicted the correct JR target.

This information is incredibly expensive to store. Instead of passing PCs down the pipeline, branch and jump instructions access the ROB's "PC File" during the *Register-read* stage for use in the Branch Unit. Two optimizations are used:

- only a single PC is stored per ROB row.[1]
- the PC File is stored in two banks, allowing a single read-port to read two consecutive entries simultaneously (for use with JR instructions).

6.3 The Commit Stage

When the instruction at the *commit head* is no longer busy (and it is not excepting), it may be committed and its changes to the architectural state of the machine made visible. For superscalar commit, the entire ROB row is analyzed for *not busy* instructions (and thus, up to the entire ROB row may be committed in a single cycle). The ROB will greedily commit as many instructions as it can per row to release resource as soon as possible. However, the ROB does not (currently) look across multiple rows to find commit-able instructions.

Only once a store has been committed may it be sent to memory. For superscalar committing of stores, the LSU is told "how many stores" may be marked as committed. The LSU will then drain the committed stores to memory as it sees fit.

When an instruction (that writes to a register) commits, it then frees the *stale physical destination register*. The *stale pdst* is then free to be re-allocated to a new instruction.

6.4 Exceptions and Flushes

Exceptions are handled when the instruction at the *commit head* is excepting. The pipeline is then flushed and the ROB emptied. The rename map tables must be reset to represent the true, non-speculative *committed* state. The front-end is then directed to the appropriate PC. If an architectural exception, this is the *exception vector* as specified by the Control/Status Registers. If it is a micro-architectural exception (e.g., a load/store ordering misspeculation) the failing instruction is refetched and execution can begin anew.

[1] Because instructions within an ROB row are consecutive in the program, the instruction's ROB bank implicitly provides the lower PC bits.

6.4.1 Parameterization - Rollback versus Single-cycle Reset

The behavior of resetting the map tables is parameterizable. The first option, as the only map table snapshots are for branches, is to rollback the ROB one row per cycle to unwind the rename state (this is the behavior of the MIPS R10k[3]). For each instruction, the *stale physical destination* register is written back into the map table for its *logical destination* specifier.

However, a faster single-cycle reset is available. This is accomplished by using another rename snapshot that tracks the *committed* state of the rename tables. This *committed map table* is updated as instructions commit.

6.4.2 Causes

The RV64G ISA provides relatively few exception sources:

Load/Store Unit - page faults and memory ordering speculation errors

Branch Unit - misaligned fetches

Decode Stage - all other exceptions and interrupts can be handled before the instruction is dispatched to the ROB

Note that memory ordering speculation errors are treated as exceptions, but actually only cause a pipeline "retry".

Chapter 7

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.

BOOM uses a "unified" issue window - all instructions of all types (integer, floating point, memory) are placed into a single issue window.

7.1 Speculative Issue

Although not yet supported, future designs may choose to speculatively issue micro-ops for improved performance (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, then all issued micro-ops that fall within the speculated window must be killed and retried from the issue window. More advanced techniques are also available.

7.2 Issue Slot

Figure 7.1 shows a single issue slot from the *Issue Window*.[1]

Instructions 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*.

7.3 Issue Select Logic

Each issue select logic port is a static-priority encoder that picks that first available micro-op in the issue window. Each port will only schedule a micro-op that its port can handle (e.g., floating point

[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.

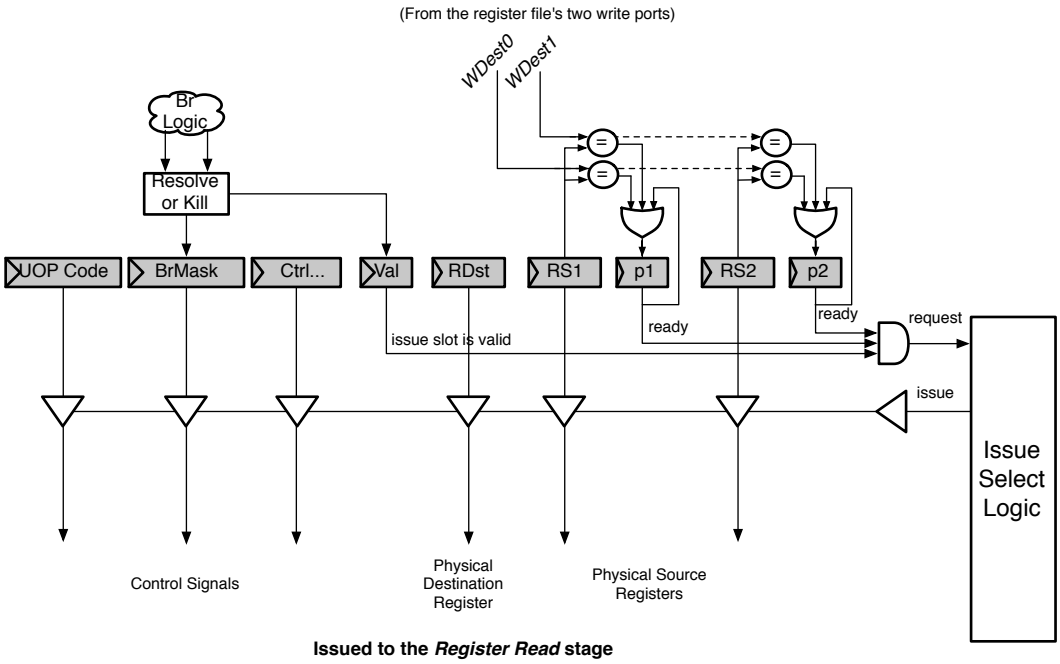

Figure 7.1: A single issue slot from the Issue Window.

micro-ops will only be scheduled onto the port governing the Floating Point Unit). This creates a cascading priority encoder for ports that can schedule the same micro-ops as each other.

If a functional unit is unavailable, it de-asserts its available signal and instructions will not be issued to it (e.g., an un-pipelined divider).

7.4 Un-ordered Issue Window

There are two scheduling policies available in BOOM.

The first is a R10K-style un-ordered issue window.[3] Dispatching instructions are placed into the first available issue window slot and remain there until they are *issued*. This can lead to pathologically poor performance, particularly in scenarios where unpredictable branches are placed into the lower priority slots and are unable to be issued until the ROB fills up and the issue window starts to drain. Because instructions following branches are only *implicitly* dependent on the branch, there is no other forcing function that enables the branches to issue earlier, except the filling of the ROB.

7.5 Age-ordered Issue Window

The second available policy is an age-ordered issue window. Dispatched instructions are placed into the bottom of the issue window (at lowest priority). Every cycle, every instruction is shifted upwards (the issue window is a "collapsing queue"). Thus, the oldest instructions will have the highest issue priority. While this increases performance by scheduling older branches and older loads

as soon as possible, it comes with a potential energy penalty as potentially every issue window slot is being read and written to on every cycle.

7.6 Wake-up

There are two types of wake-up in BOOM - *fast* wakeup and *slow* wakeup. Because ALU micro-ops can send their write-back data through the bypass network, issued ALU micro-ops will broadcast their wakeup to the issue-window as they are issued.

However, floating-point operations, loads, and variable latency operations are not sent through the bypass network, and instead the wakeup signal comes from the register file ports during the *write-back* stage.

Chapter 8

The Register File and Bypass Network

BOOM is a unified, physical register file (PRF) design. The register file holds both the committed and speculative state. The register file also holds both integer and floating point register values. The map tables track which physical register corresponds to which ISA register.

BOOM uses the Berkeley hardfloat floating point units which use an internal 65-bit operand format (https://github.com/ucb-bar/berkeley-hardfloat). Therefore, all physical registers are 65-bits.

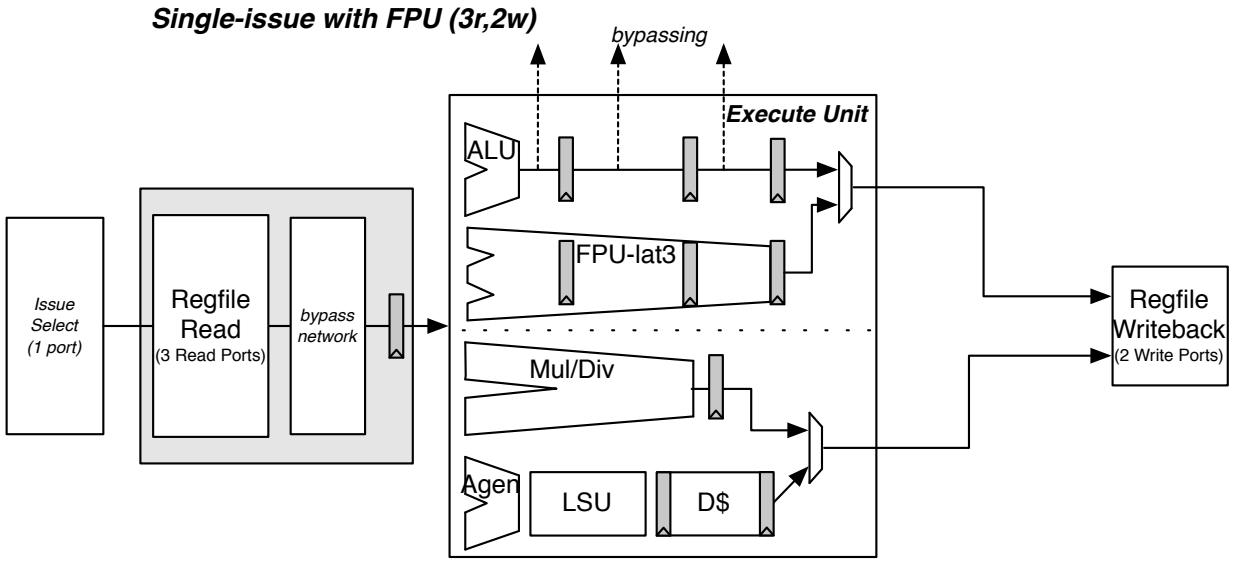

Figure 8.1: An example single-issue pipeline. The register file needs 3 read ports to satisfy FMA operations and 2 write ports to satisfy the variable latency operations without interfering with the fixed latency ALU and FP write-backs. To make scheduling of the write port trivial, the ALU's pipeline is lengthened to match the FPU latency. The ALU is able to bypass from any of these stages to dependent instructions in the *Register Read* stage.

8.1 Register Read

The register file statically provisions all of the register read ports required to satisfy all issued instructions. For example, if *issue port #0* corresponds to an integer ALU and *issue port #1* corresponds to a FPU, then the first two register read ports will statically serve the ALU and the next three register read ports will service the FPU for five total read ports.

8.1.1 Dynamic Read Port Scheduling

Future designs can improve area-efficiency by provisioning fewer register read ports and using dynamically scheduling to arbitrate for them. This is particularly helpful as most instructions need only one operand. However, it does add extra complexity to the design, which is often manifested as extra pipeline stages to arbitrate and detect structural hazards. It also requires the ability to kill issued micro-ops and re-issue them from the issue window on a later cycle.

8.2 Bypass Network

ALU operations can be issued back-to-back by having the write-back values forwarded through the bypass network. Bypassing occurs at the end of the *Register Read* stage.

Chapter 9

The Execute Pipeline

The Execution Pipeline covers the execution and write-back of micro-ops. Although the micro-ops will travel down the pipeline one after the other (in the order they have been issued), the micro-ops themselves are likely to have been issued to the Execution Pipeline out-of-order. Figure 9.1 shows an example Execution Pipeline for a dual-issue BOOM.

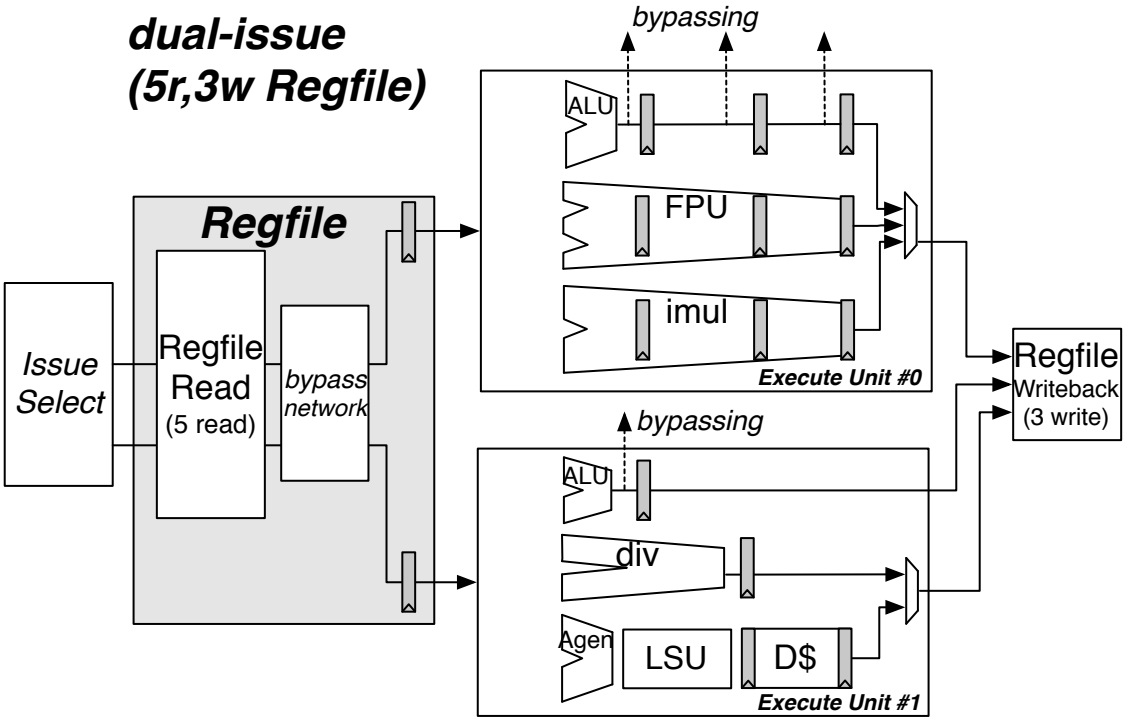

Figure 9.1: An example pipeline for a dual-issue BOOM. The first issue port schedules micro-ops onto Execute Unit #0, which can accept ALU operations, FPU operations, and integer multiply instructions. The second issue port schedules ALU operations, integer divide instructions (unpipelined), and load/store operations. The ALU operations can bypass to dependent instructions. Note that the ALU in EU#0 is padded with pipeline registers to match latencies with the FPU and iMul units to make scheduling for the write-port trivial. Each Execution Unit has a single issue-port dedicated to it but contains within it a number of lower-level Functional Units.

9.1 Execution Units

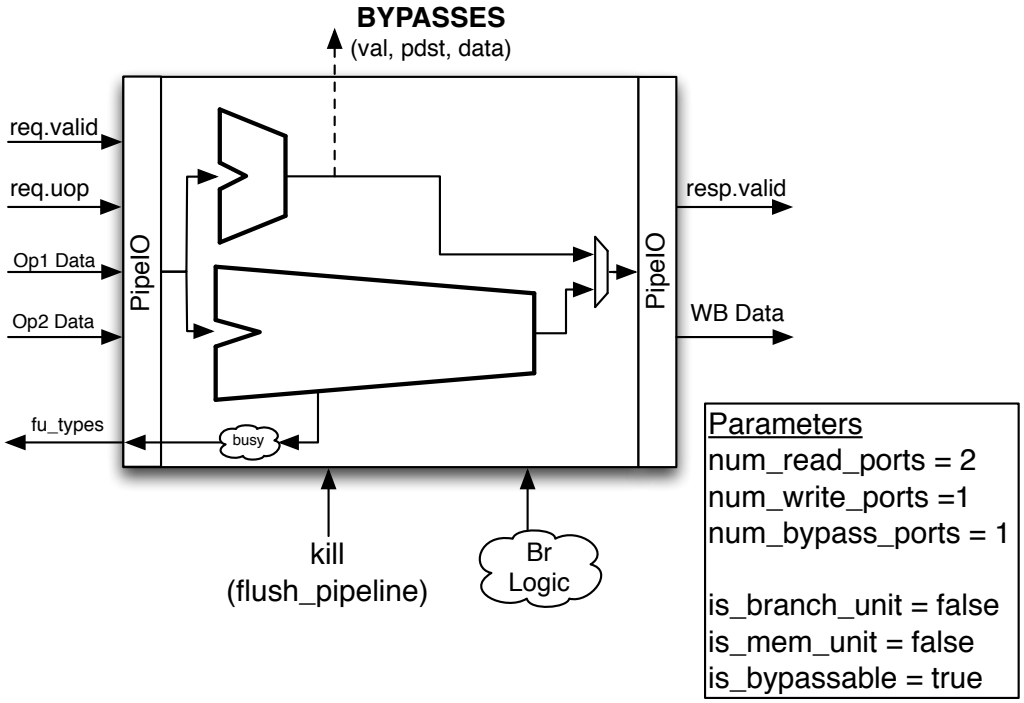

Figure 9.2: An example Execution Unit. This particular example shows an integer ALU (that can bypass results to dependent instructions) and an unpipelined divider that becomes *busy* during operation. Both functional units share a single write-port. The Execution Unit accepts both *kill* signals and *branch resolution* signals and passes them to the internal functional units as required.

An Execution Unit is a module that a single issue port will schedule micro-ops onto and contains some mix of functional units. Phrased in another way, each issue port from the Issue Window talks to one and only one Execution Unit. An Execution Unit may contain just a single simple integer ALU, or it could contain a full complement of floating point units, a integer ALU, and an integer multiply unit.

The purpose of the Execution Unit is to provide a flexible abstraction which gives a lot of control over what kind of Execution Units the architect can add to their pipeline

9.1.1 Scheduling Readiness

An Execution Unit provides a bit-vector of the functional units it has available to the issue scheduler. The issue scheduler will only schedule micro-ops that the Execution Unit supports. For functional units that may not always be ready (e.g., an un-pipelined divider), the appropriate bit in the bit-vector will be disabled (See Fig 9.2).

9.2 Functional Units

Functional units are the muscle of the CPU, computing the necessary operations as required by the instructions. Functional units typically require a knowledgable domain expert to implement them correctly and efficiently.

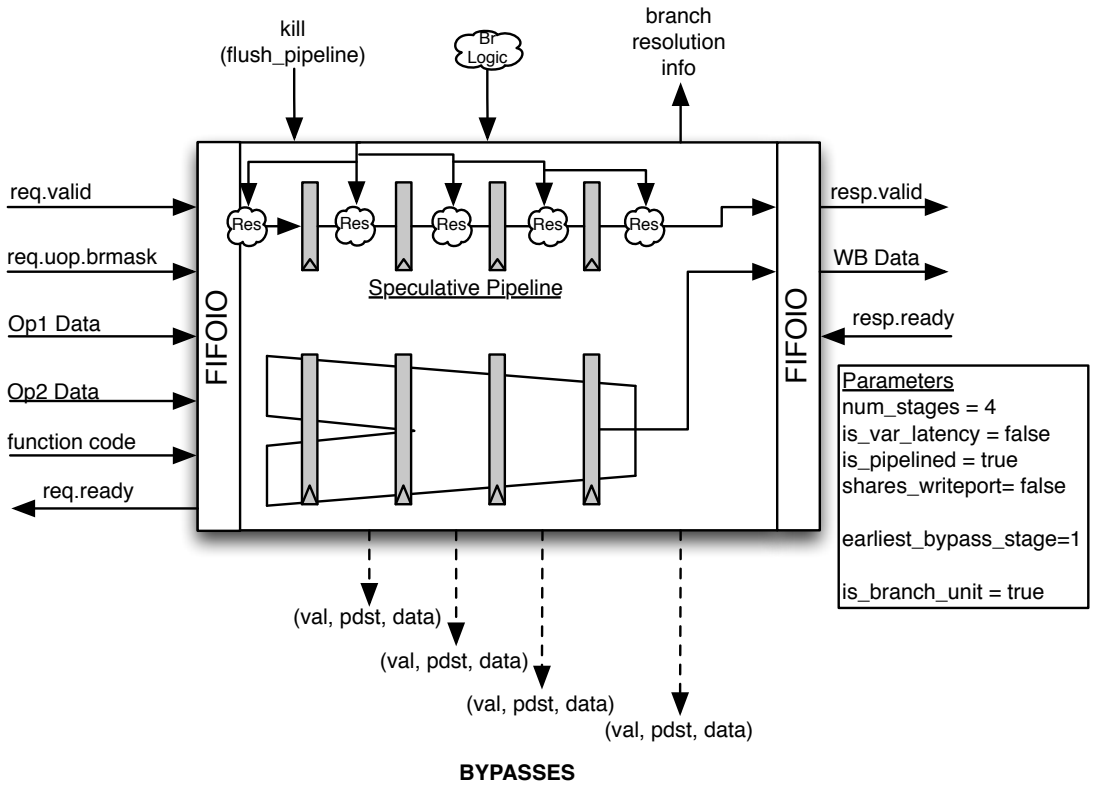

Figure 9.3: The abstract Pipelined Functional Unit class. An expert-written, low-level functional unit is instantiated within the Functional Unit. The request and response ports are abstracted and bypass and branch speculation support is provided. Micro-ops are individually killed by gating off their response as they exit the low-level functional unit.

For this reason, BOOM uses an abstract Functional Unit class to "wrap" expert-written, low-level functional units from the Rocket repository (see Section 1.6.1). However, the expert-written functional units created for the Rocket in-order processor make assumptions about in-order issue and commit points (namely, that once an instruction has been dispatched to them it will never need to be killed). These assumptions break down for BOOM.

However, instead of re-writing or forking the functional units, BOOM provides an abstract Functional Unit class (see Fig 9.3) that "wraps" the lower-level functional units with the parameterized auto-generated support code needed to make them work within BOOM. The request and response ports are abstracted, allowing Functional Units to provide a unified, interchangeable interface.

9.2.1 Pipelined Functional Units

A pipelined functional unit can accept a new micro-op every cycle. Each micro-op will take a known, fixed latency.

Speculation support is provided by auto-generating a pipeline that passes down the micro-op meta-data and *branch mask* in parallel with the micro-op within the expert-written functional unit. If a micro-op is misspeculated, it's response is de-asserted as it exits the functional unit.

An example pipelined functional unit is shown in Fig 9.3.

9.2.2 Un-pipelined Functional Units

Un-pipelined functional units (e.g., a divider) take an variable (and unknown) number of cycles to complete a single operation. Once occupied, they de-assert their ready signal and no additional micro-ops may be scheduled to them.

Speculation support is provided by tracking the *branch mask* of the micro-op in the functional unit.

The only requirement of the expert-written un-pipelined functional unit is to provide a *kill* signal to quickly remove misspeculated micro-ops.

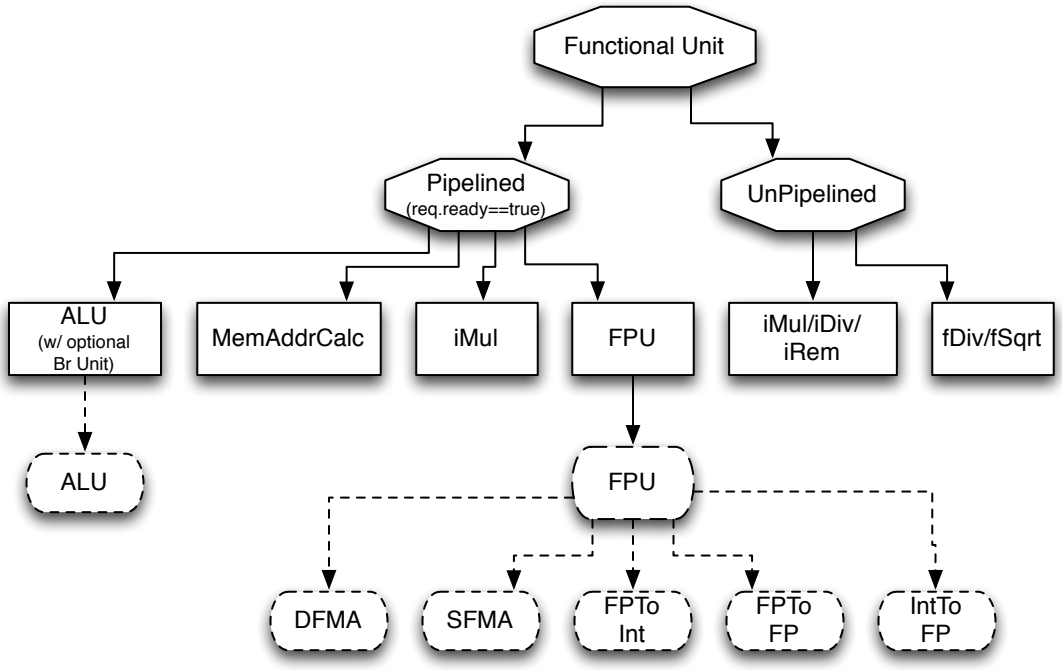

Figure 9.4: The dashed ovals are the low-level functional units written by experts, the squares are concrete classes that instantiate the low-level functional units, and the octagons are abstract classes that provide generic speculation support and interfacing with the BOOM pipeline.

9.3 Branch Unit & Branch Speculation

The Branch Unit handles the resolution of all branch and jump instructions.

All micro-ops that are "inflight" in the pipeline (have an allocated ROB entry) are given a *branch mask*, where each bit in the *branch mask* corresponds to an un-executed, inflight branch that the micro-op is speculated under. Each branch in *Decode* is allocated a *branch tag*, and all following micro-ops will have the corresponding bit in the *branch mask* set (until the branch is resolved by the Branch Unit).

If the branches (or jumps) have been correctly speculated by the front-end, then the Branch Unit's only action is to broadcast the corresponding branch tag to *all* inflight micro-ops that the branch has been resolved correctly. Each micro-op can then clear the corresponding bit in its *branch mask*, and that branch tag can then be allocated to a new branch in the *Decode* stage.

If a branch (or jump) is misspeculated, the Branch Unit must redirect the PC to the correct target, kill the front-end and fetch buffer, and broadcast the misspeculated *branch tag* so that all dependent, inflight micro-ops may be killed. The PC redirect signal goes out immediately, to decrease the misprediction penalty. However, the *kill* signal is delayed a cycle for critical path reasons.

The front-end must pass down the pipeline the appropriate branch speculation meta-data, so that the correct direction can be reconciled with the prediction. Jump Register instructions are evaluated by comparing the correct target with the PC of the next instruction in the ROB (if not available, then a misprediction is assumed). Jumps are evaluated and handled in the front-end (as their direction and target are both known once the instruction can be decoded).

BOOM (currently) only supports having one Branch Unit.

9.4 Load/Store Unit

.

The Load/Store Unit (LSU) handles the execution of load, store, atomic, and fence operations.

BOOM (currently) only supports having one LSU (and thus can only send one load or store per cycle to memory).[1]

See Chapter 10 for more details on the LSU.

9.5 Floating Point Units

The low-level floating point units used by BOOM come from the Rocket processor (https://github.com/ucb-bar/rocket) and hardfloat (https://github.com/ucb-bar/berkeley-hardfloat) repositories. Figure 9.5 shows the class hierarchy of the FPU.

To make the scheduling of the write-port trivial, all of the FP units are padded to have the same latency.[2]

[1] Relaxing this constraint could be achieved by allowing multiple LSUs to talk to their own bank(s) of the data-cache, but the added complexity comes in allocating entries in the LSU before knowing the address, and thus which bank, a particular memory operation pertains to.

[2] Rocket instead handles write-port scheduling by killing and refetching the offending instruction (and all instructions behind it) if there is a write-port hazard detected. This would be far more heavy-handed to do in BOOM.

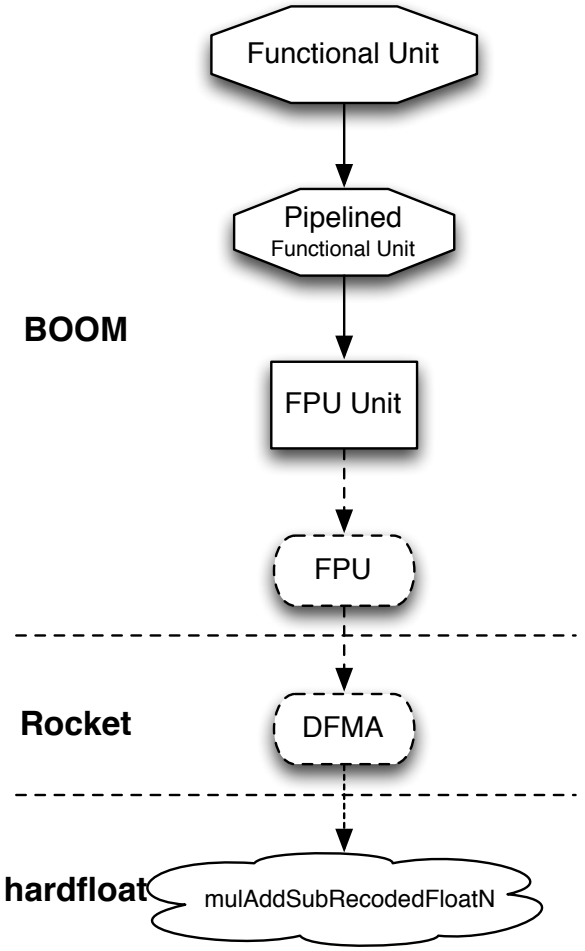

Figure 9.5: The class hierarchy of the FPU is shown. The expert-written code is contained within the `hardfloat` and `rocket` repositories. The "FPU" class instantiates the Rocket components, which itself is further wrapped by the abstract Functional Unit classes (which provides the out-of-order speculation support).

9.6 Parameterization

BOOM provides flexibility in specifying the issue width and the mix of functional units in the execution pipeline. Code 9.6 shows how to instantiate an execution pipeline in BOOM.

```
1  val exe_units = ArrayBuffer[ExecutionUnit]()
2
3  if (ISSUE_WIDTH == 2)
4  {
5     exe_units += Module(new ALUExeUnit(is_branch_unit = true
6                                        , has_mul     = true
7                                        ))
8     exe_units += Module(new ALUMemExeUnit(has_div     = true
9                                        ))
10 }
11 else if (ISSUE_WIDTH == 3)
12 {
13    exe_units += Module(new ALUExeUnit(is_branch_unit = true
14                                       , has_mul     = true
15                                       ))
16    exe_units += Module(new ALUExeUnit(has_div = true))
17    exe_units += Module(new MemExeUnit())
18 }
```

Code 9.1: Instantiating the Execution Pipeline (in dpath.scala). Adding execution units is as simple as instantiating another ExecutionUnit module and adding it to the exe_units ArrayBuffer.

Additional parameterization, regarding things like the latency of the FP units can be found within the Configuration settings (configs.scala).

9.7 Control/Status Register Instructions

A set of Control/Status Register (CSR) instructions allow the atomic read and write of the Control/Status Registers. These architectural registers are separate from the integer and floating registers, and include the cycle count, retired instruction count, status, exception PC, and exception vector registers (and many more!). Each CSR has its own required privilege levels to read and write to it and some have their own side-effects upon reading (or writing).

BOOM (currently) does not rename *any* of the CSRs, and in addition to the potential side-effects caused by reading or writing a CSR, **BOOM will only execute a CSR instruction non-speculatively.**[3] This is accomplished by marking the CSR instruction as a "unique" (or "serializing") instruction - the ROB must be empty before it may proceed to the Issue Window (and no instruction may follow it until it has finished execution and been committed by the ROB). It is then issued by the Issue Window, reads the appropriate operands from the Physical Register File, and is then sent to the CSRFile.[4] The CSR instruction executes in the CSRFile and then writes back data as required to the Physical Register File. The CSRFile may also emit a PC redirect and/or an exception as part of executing a CSR instruction (e.g., a syscall).

[3]There is a lot of room to play regarding the CSRs. For example, it is probably a good idea to rename the sscratch register (dedicated for use by the supervisor) as it may see a lot of use in some kernel code and it causes no side-effects.

[4]The CSRFile is a Rocket component.

Chapter 10

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 10.0.2 for more details on the store micro-op specifics.

10.0.1 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.

10.0.2 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.

10.0.3 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 retried at a later time.[2]

10.0.4 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 10.1 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.

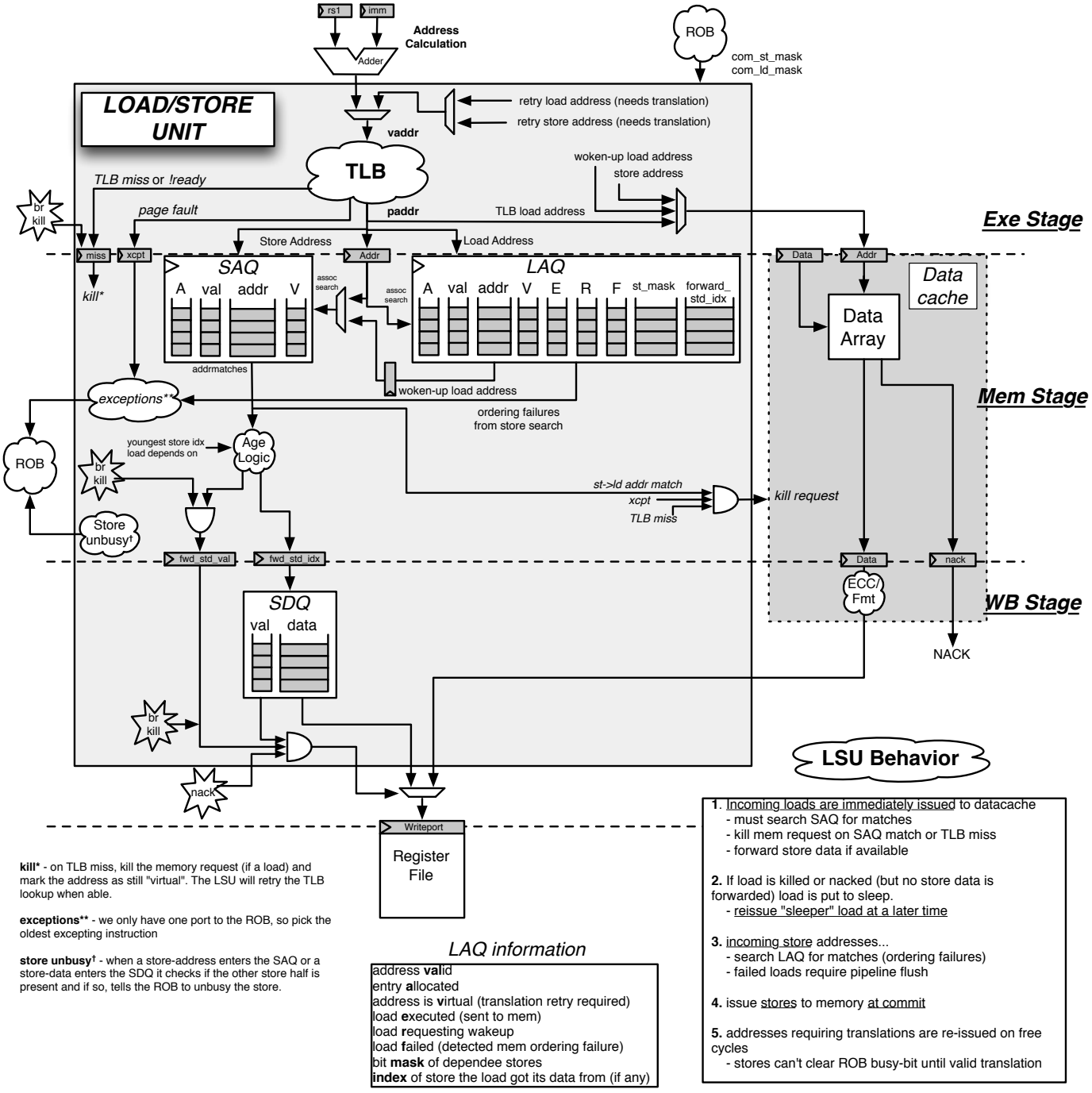

Figure 10.1: The Load/Store Unit.

Chapter 11

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 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 lie in the vector load/store unit.

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.

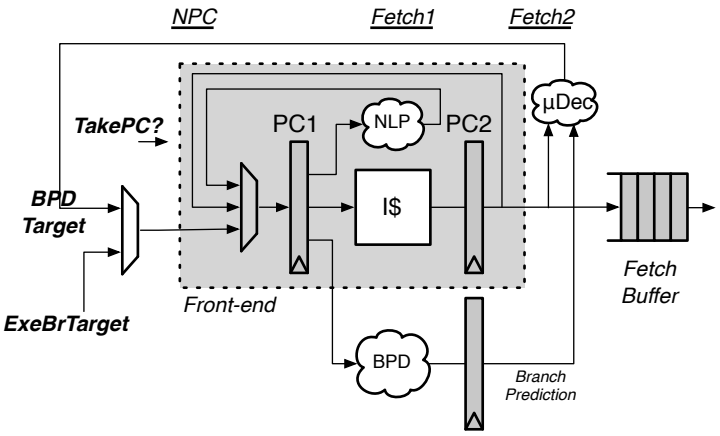

Figure A.1: The Fetch Unit. The grey box encompasses the Rocket front-end, which is re-used by BOOM.

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, a few include:

- 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 strategy of RVC in BOOM:

- Implement RVC in the Rocket in-order core. Done properly, BOOM may 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.

A.3.1 Challenging Implementation Details

There are many challenging corner cases to consider with adding RVC support to BOOM. First, although all 16 bit encodings map to a 32b version, **the behavior of some 16b instructions are different from their 32b counterparts**! A JAL instruction writes the address of the following instruction to `rd` - but whether that is $PC + 2$ or $PC + 4$ depends on whether it's the 16b JAL or a 32b JAL! Likewise, a mispredicted not-taken branch redirects the fetch unit to $PC + 2$ or $PC + 4$ depending on whether the branch was the compressed version or not. **Thus, the pipeline must track whether any given instruction was originally a compressed 16b instruction or not.**

The branch prediction units will also require a careful rethink. The BTB tracks which instructions are *predicted-taken* branches and redirects the PC as desired. For a superscalar *fetch packet*, the BTB must help denote which instruction is to be blamed for the taken prediction to help mask off any invalid instructions that come afterward within the *fetch packet*. RVC makes this much more difficult, as some *predicted-taken* branches can wrap around fetch groupings/cache lines/virtual page boundaries. Thus, the "taken" prediction must be attached to a tag-hit on the *end* of the branch instruction. This handles fetching the first part of the branch (and predicting "not-taken"), then fetching the second part (which hits in the BTB and predicts "taken"), and only then redirecting the front-end to the predicted-taken PC target.

Appendix B

Parameterization

B.1 Rocket Parameters

B.2 BOOM Parameters

B.3 Uncore Parameters

Appendix C

Frequently Asked Questions

To be filled in as questions are asked!

Appendix D

Terminology

[TODO: To be filled in as needed.]
[TODO: come up with a better format...]

fetch packet - A bundle returned by the front-end which contains some set of consecutive instructions with a mask denoting which instructions are valid, amongst other meta-data related to instruction fetch and branch prediction.

fetch PC - The PC corresponding to the beginning of a *fetch packet*.

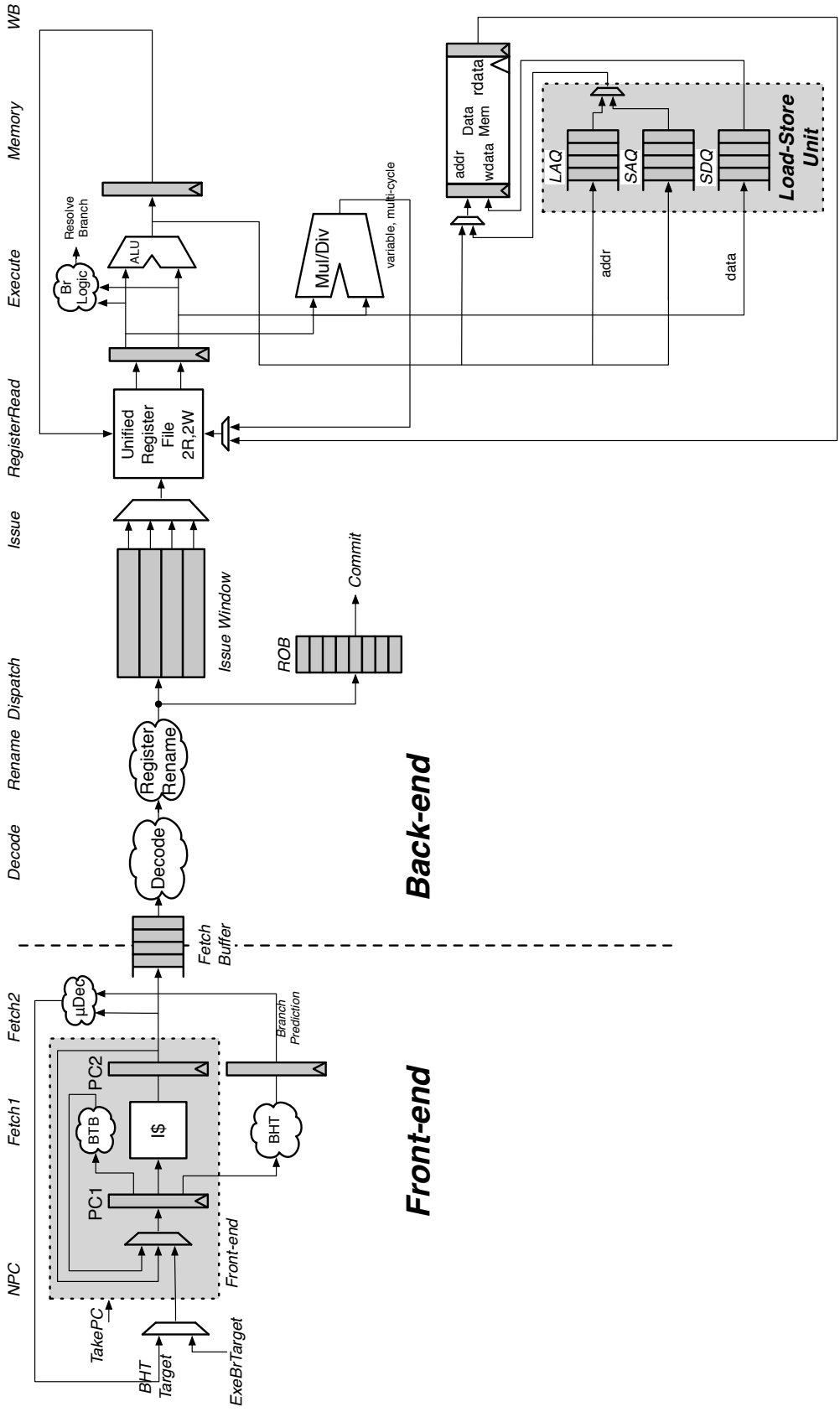

Figure D.1: A more detailed diagram of BOOM, for a single-issue RV32IM design.

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.