RISCV-BOOM Documentation

Chris Celio, Jerry Zhao, Abraham Gonzalez, Ben Korpan

Dec 23, 2018

Contents:

CHAPTER 1

Introduction and 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. 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://github.com/riscv-boom/riscv-boom.

1.1 The BOOM Pipeline

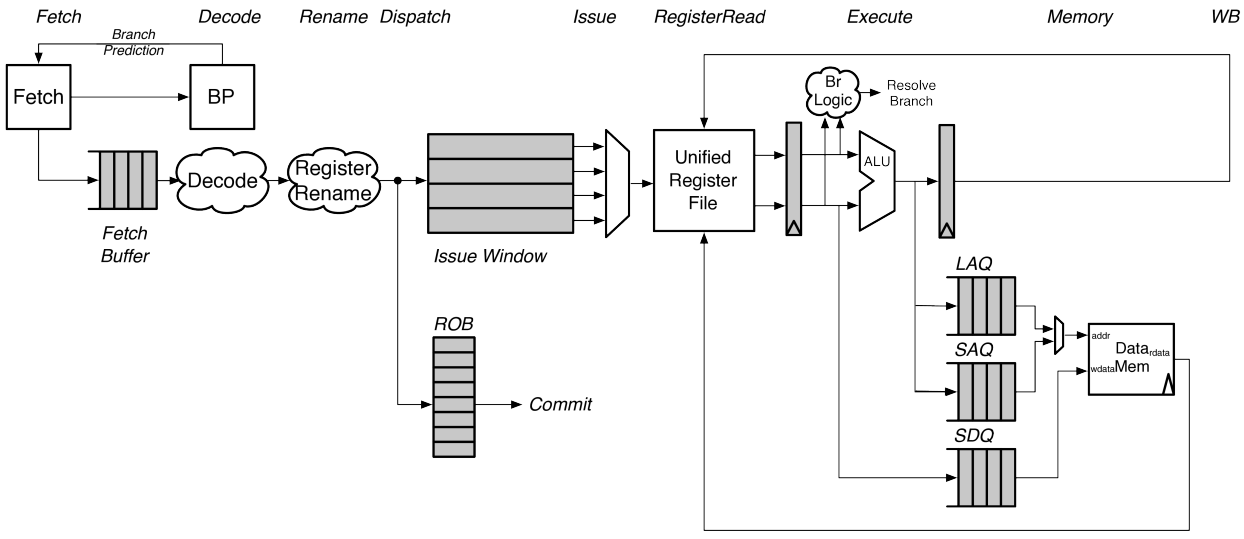


Fig. 1.1: BOOM Pipeline Stages

1.1.1 Overview

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").

1.1.2 Stages

Fetch

Instructions are **fetched** from the Instruction Memory and pushed 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 micro-op is then **dispatched**, or written, into the **Issue Window**.

Issue

Micro-ops 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 micro-ops 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.

[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 [chapter:memory].

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

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 LAQ. Stores are fired to memory at **Commit** time (and naturally, stores cannot be **committed** until both their address and data have been placed in the SAQ and SDQ).

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, the ROB **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.

1.1.3 Branch Support

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 Fig. 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.

1.2 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 . 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 hardware construction language. More information about can be found at http://chisel.eecs.berkeley.edu.

1.4 Quick-start

The best way to get started with the BOOM core is to use the BOOM project template located in the main GitHub organization. There you will find the main steps to setup your environment, build, and run the BOOM core on a C++ emulator. Here is a selected set of steps from that repositories README:

Listing 1.1: Quick-Start Code

```
# Download the template and setup environment
git clone https://github.com/riscv-boom/boom-template.git
cd boom-template
./scripts/init-submodules.sh

# You may want to add the following two lines to your shell profile
export RISCV=/path/to/install/dir
export PATH=$RISCV/bin:$PATH

cd boom-template
./scripts/build-tools.sh

cd verisim
make run
```

Note: Listing 1.1 assumes you have don't have installed the riscv-tools toolchain. It will pull and build the toolchain for you.

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/freechipsproject/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/
  - bpu/
    - 2bc-table.scala
    - base-only.scala
    - bim.scala
    - bpd-pipeline.scala
    - brpredictor.scala
    - btb-sa.scala
    - btb.scala
    - dense-btb.scala
    - gshare.scala
    - tage.scala
    - tage-table.scala
  - common/
    - configs
    - consts
    - microop
    - package
    - parameters
    - tile
    - types
  - exu/
    - core.scala
    - decode.scala
    - execute.scala
    - execution_units.scala
    - fdiv.scala
    - fppipeline.scala
    - fpu.scala
    - fudecode.scala
    - functional_unit.scala
    - imul.scala
    - issue_ageordered.scala
    - issue.scala
    - issue_slot.scala
    - issue_unordered.scala

        * regfile-custom.scala
        * regfile.scala
        * registerread.scala
        * rename-busytable.scala
        * rename-freelist.scala
        * rename-maptable.scala
        * rename.scala
        * rob.scala
    - ifu/
        * branchchecker.scala
        * fetchbuffer.scala
        * fetchmonitor.scala
        * fetch.scala
        * fetchtargetqueue.scala
        * frontend.scala
        * icache.scala
    - lsu/
        * dcacheshim.scala
        * lsu.scala
        * types.scala
    - system/
        * BoomSubsystem.scala
        * BoomTestSuites.scala
        * Configs.scala
        * ExampleBoomSystem.scala
        * Generator.scala
        * TestHarness.scala
    - util/
        * elastic-reg.scala
        * elastic-sram.scala
        * seqmem-transformable.scala
        * util.scala

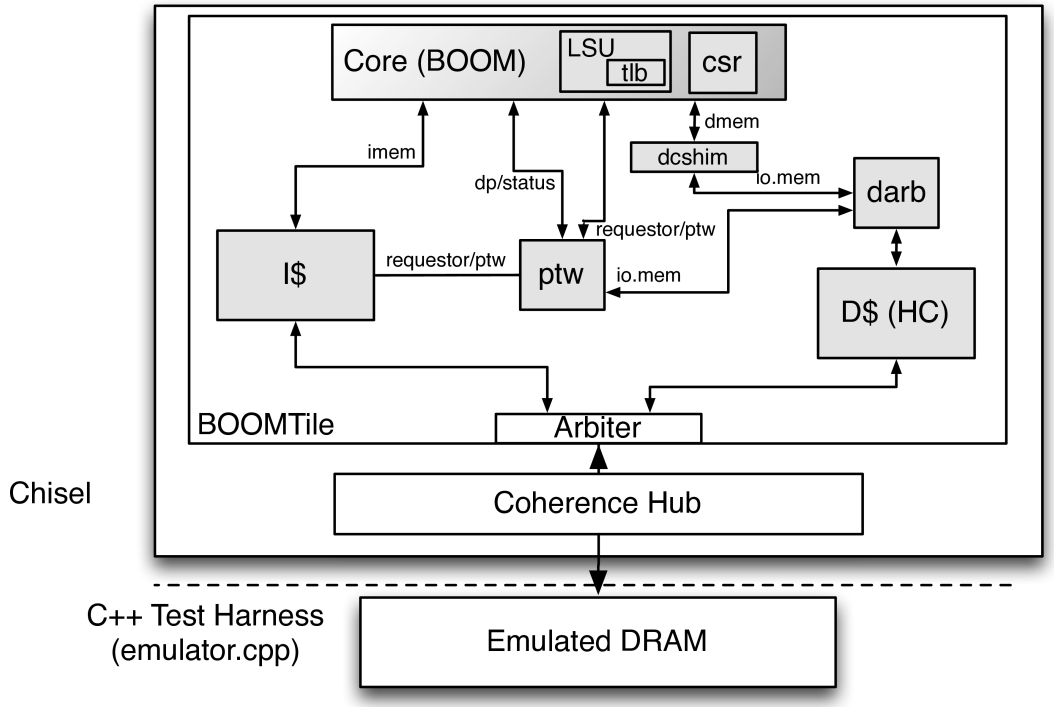


Fig. 1.2: A single-core "BOOM-chip", with no L2 last-level cache

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/freechipsproject/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.

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 source code for the BOOM core.**
  - chisel **The source code to the Chisel language itself.**
  - firrtl **The source code to the FIRRTL project.**
  - csrc/ **Utility C/C++ source code.**
  - emulator/ **Verilator simulation tools and support.**
  - generated-src/ **Auto-generated Verilog code.**
  - Makefile **Makefile for Verilator simulation.**
  - output/ **Output files from Verilator simulation runs.**
  - 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/ **source code for rocket-chip.**
- rocket/ **Git submodule of the source code for the Rocket core (used as a library of processor components).**
- junctions/ **Git submodule of the source code for the uncore and off-chip network.**
- uncore/ **Git submodule of the source code for the uncore components (including LLC).**
- sbt/ **Scala voodoo.**
- vsim/ **The ASIC Verilog simulation and build directories.**

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 thirteen times in three 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/freechipsproject/rocket-chip.

CHAPTER 2

Instruction Fetch

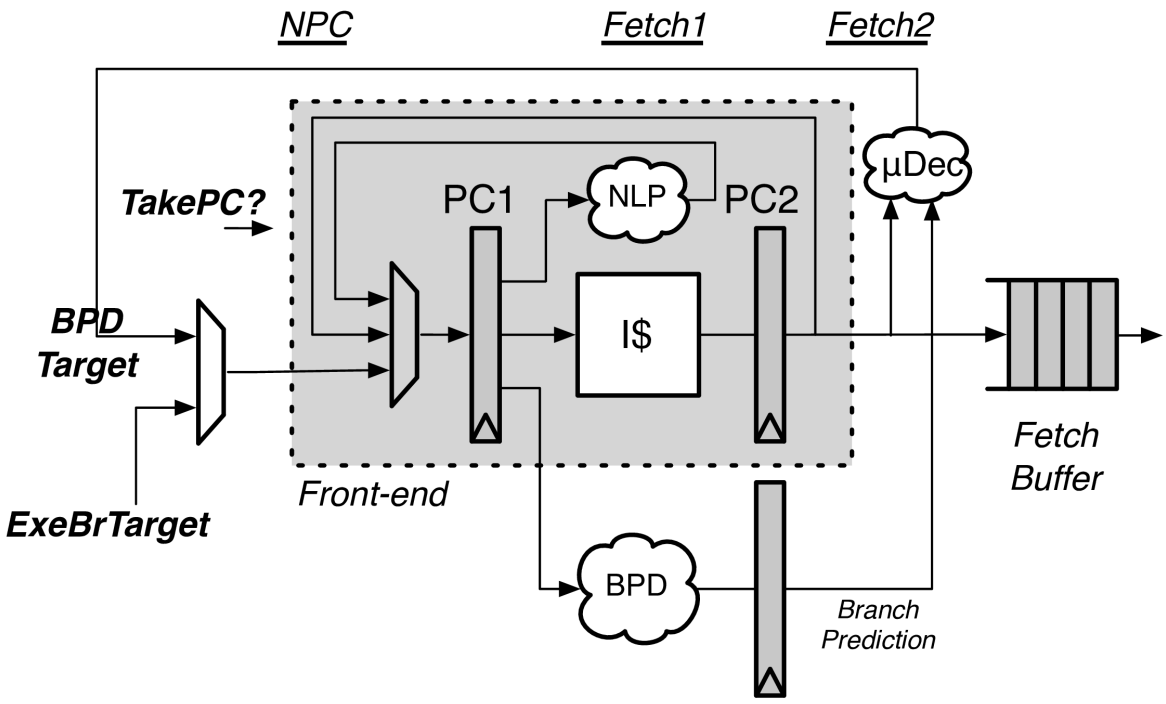


Fig. 2.1: The Fetch Unit. The grey box is the front-end instantiated from the Rocket code base.

BOOM instantiates the Rocket core's *Front-end* (highlighted in grey in Fig. 2.1 , 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 *Branch Prediction* for more

information on how branch prediction fits into the Fetch Unit's pipeline.

Since 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

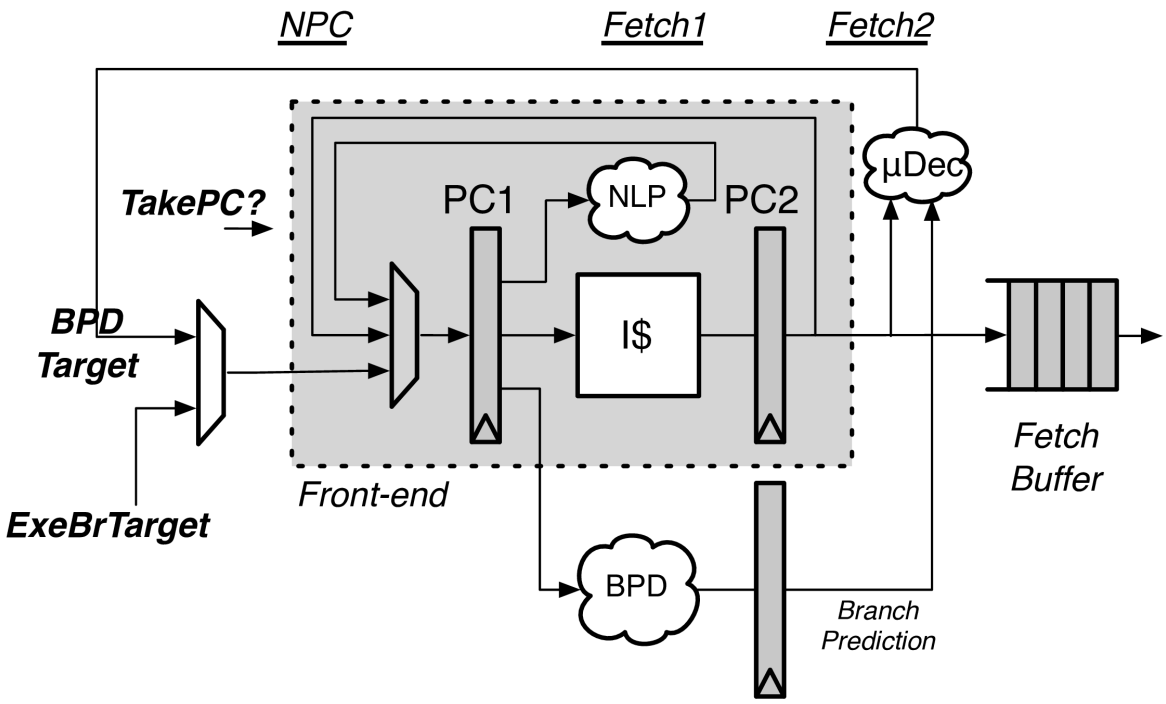


Fig. 3.1: The Fetch Unit

This chapter discusses how BOOM predicts branches and then resolves these predictions.

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

[1] Unfortunately, the terminology in the literature gets a bit muddled here in what to call different types and levels of branch predictor. I have

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 backing 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.

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) which work together to make a fast, but reasonably accurate prediction.

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.

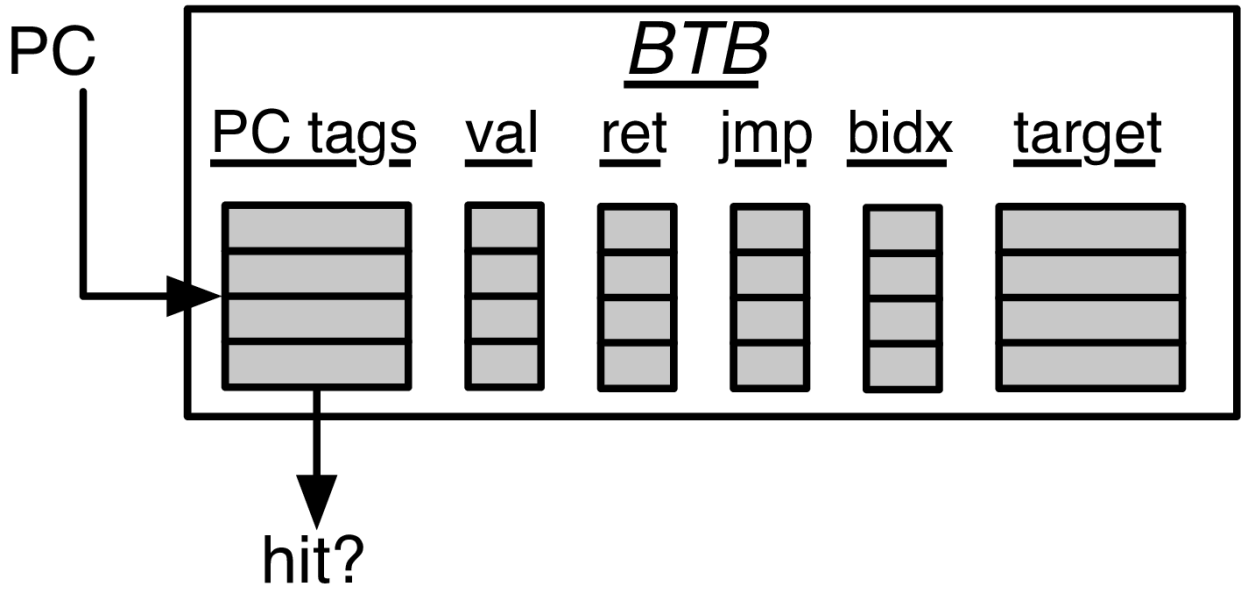


Fig. 3.2: The Next-line Predictor (NLP) Unit. The Fetch PC scans the BTB's "PC tags" for a match. If a match is found (and the entry is valid), the BHT and RAS are consulted for the final verdict. If the entry is a "ret" (return instruction), then the target comes from the RAS. If the entry is a unconditional "jmp" (jump instruction), then the BHT is not consulted. The "bidx", or branch index, marks which instruction in a superscalar fetch packet is the cause of the control flow prediction. This is necessary to mask off the other instructions in the fetch packet that come over the taken branch

seen "micro-BTB" versus "BTB", "NLP" versus "BHT", and "cache-line predictor" versus "overriding predictor". Although the Rocket code calls its own predictor the "BTB", I have chosen to refer to it in documentation as the "next-line predictor", to denote that it is a combinational predictor that provides single-cycle predictions for fetching "the next line", and the Rocket BTB encompasses far more complexity than just a "branch target buffer" structure. Likewise, I have chosen the name "backing predictor" as I believe it is the most accurate name, while simultaneously avoiding being overly descriptive of the internal design (is it a simple BHT? Is it tagged? Does it override the NLP?). But in short, I am open to better names!

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 (of when to or the stack) 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 also its *Fetch PC* (the PC of the head instruction of its *fetch packet*.[2]

BTB Updates

The BTB is updated **only** when the Fetch Unit is redirected to **take** a branch or jump by either the Branch Unit (in the *Execute* stage) or the Backing Predictor (in the *Branch Predict* stage).[3]

If there is no BTB entry corresponding to the taken branch or jump, an new entry is allocated for it.

BHT Updates

The BHT is composed of two parts that require updates - a *global history (ghistory)* register and a table of *history counters*.

The register tracks the outcomes of the last *N* branches that have been fetched. It must be updated:

- in the *Branch Predict* stage - once we have decoded the instruction *fetch bundle*, know if any branches are present, and which direction the branch predictors have chosen to direct the Fetch Unit.
- in the *Execute* stage - if and only if a *misprediction* occurs, the register must be reset with the correct outcome of the branch history.

The *history counter* table is updated when the register is updated. Because the counters are read out and passed down the pipeline with the branch instruction, there is not a problem with having updated the counters incorrectly in the earlier *Branch Predict* stage. If a misprediction occurs, the counters will be reset and incremented to the proper value.

Notice that by updating the history counters in the *Branch Predict* stage, the updates are being performed in-order! However, it is possible for a branch to update the *history counters* before later being found to have been misspeculated under a previous branch. We suspect that this is a fairly benign scenario.[4]

RAS Updates

The RAS is updated during the *Branch Predict* stage once the instructions in the *fetch packet* have been decoded. If the taken instruction is a call[5] , the *Return Address* is onto the RAS. If the taken instruction is a , then the RAS is .

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

[3] Rocket's BTB relies on a little cleverness - when redirecting the PC on a misprediction, this new *Fetch PC* is the same as the *Update PC* that needs to be written into a new BTB entry's *Target PC* field. This "coincidence" allows the PC compression table to use a single search port - it is simultaneously reading the table for the next prediction while also seeing if the new *Update PC* already has the proper high-order bits allocated for it.

[4] Likewise, the BHT does not keep track of a *commit copy* of the register. This means that any sort of exceptions or pipeline replays will leave the register in an incoherent state. However, experiments showed that this had no noticeable effect on performance on real benchmarks. This is probably because the BHT's register is fairly small and can quickly re-learn the history in only a few cycles.

[5] While RISC-V does not have a dedicated instruction, it can be inferred by checking for a or instruction with a writeback destination to (aka, the).

Superscalar Predictions

When the NLP makes a prediction, it is actually using the BTB to tag match against the predicted branch's *Fetch PC*, and not the PC of the branch itself. The NLP must predict across the entire *fetch packet* which of the many possible branches will be the dominating branch that redirects the PC. For this reason, we use a given branch's *Fetch PC* rather than its own PC in the BTB tag match.[6]

3.2 The Backing Predictor (BPD)

When the next-line predictor (NLP) is predicting well, the processor's backend is provided an unbroken stream of instructions to execute. The NLP is able to provide fast, single-cycle predictions by being expensive (in terms of both area and power), very small (only a few dozen branches can be remembered), and very simple (the *gshare* hysterisis bits are not able to learn very complicated or long history patterns).

To capture more branches and more complicated branching behaviors, BOOM provides support for a "Backing Predictor", or BPD (see Fig. 3.3).

The BPD's goal is to provide very high accuracy in a (hopefully) dense area. To make this possible, the BPD will not make a prediction until the *fetch packet* has been decoded and the branch targets computed directly from the instructions themselves. This saves on needing to store the *PC tags* and *branch targets* within the BPD[7].

The BPD is accessed in parallel with the instruction cache access (see Fig. 2.1). This allows the BPD to be stored in sequential memory (i.e., SRAM instead of flip-flops). With some clever architecting, the BPD can be stored in single-ported SRAM to achieve the density desired.

3.2.1 Making Predictions

When making a prediction, the backing predictor must provide the following:

- is a prediction being made?
- a bit-vector of taken/not-taken predictions

As per the first bullet-point, the BPD may decide to not make a prediction. This may be because the predictor uses tags to inform whether its prediction is valid or there may be a structural hazard that prevented a prediction from being made.

The BPD provides a bit-vector of taken/not-taken predictions, the size of the bit-vector matching the *fetch width* of the pipeline (one bit for each instruction in the *fetch packet*). The *Branch Prediction* stage will decode the instructions in the *fetch packet*, compute the branch targets, and decide in conjunction with the BPD's prediction bit-vector if a front-end redirect should be made.

3.2.2 Jump and Jump-Register Instructions

The BPD makes predictions only on the direction (taken versus not-taken) of conditional branches. Non-conditional "jumps" and "jump-register" instructions are handled separately from the BPD.[8]

[6] Each BTB entry corresponds to a single *Fetch PC*, but it is helping to predict across an entire *fetch packet*. However, the BTB entry can only store meta-data and target-data on a single control-flow instruction. While there are certainly pathological cases that can harm performance with this design, the assumption is that there is a correlation between which branch in a *fetch packet* is the dominating branch relative to the *Fetch PC*, and - at least for narrow fetch designs - evaluations of this design has shown it is very complexity-friendly with no noticeable loss in performance. Some other designs instead choose to provide a whole bank of BTBs for each possible instruction in the *fetch packet*.

[7] It's the *PC tag* storage and *branch target* storage that makes the BTB within the NLP so expensive.

[8] instructions jump to a location, whereas instructions jump to a location.

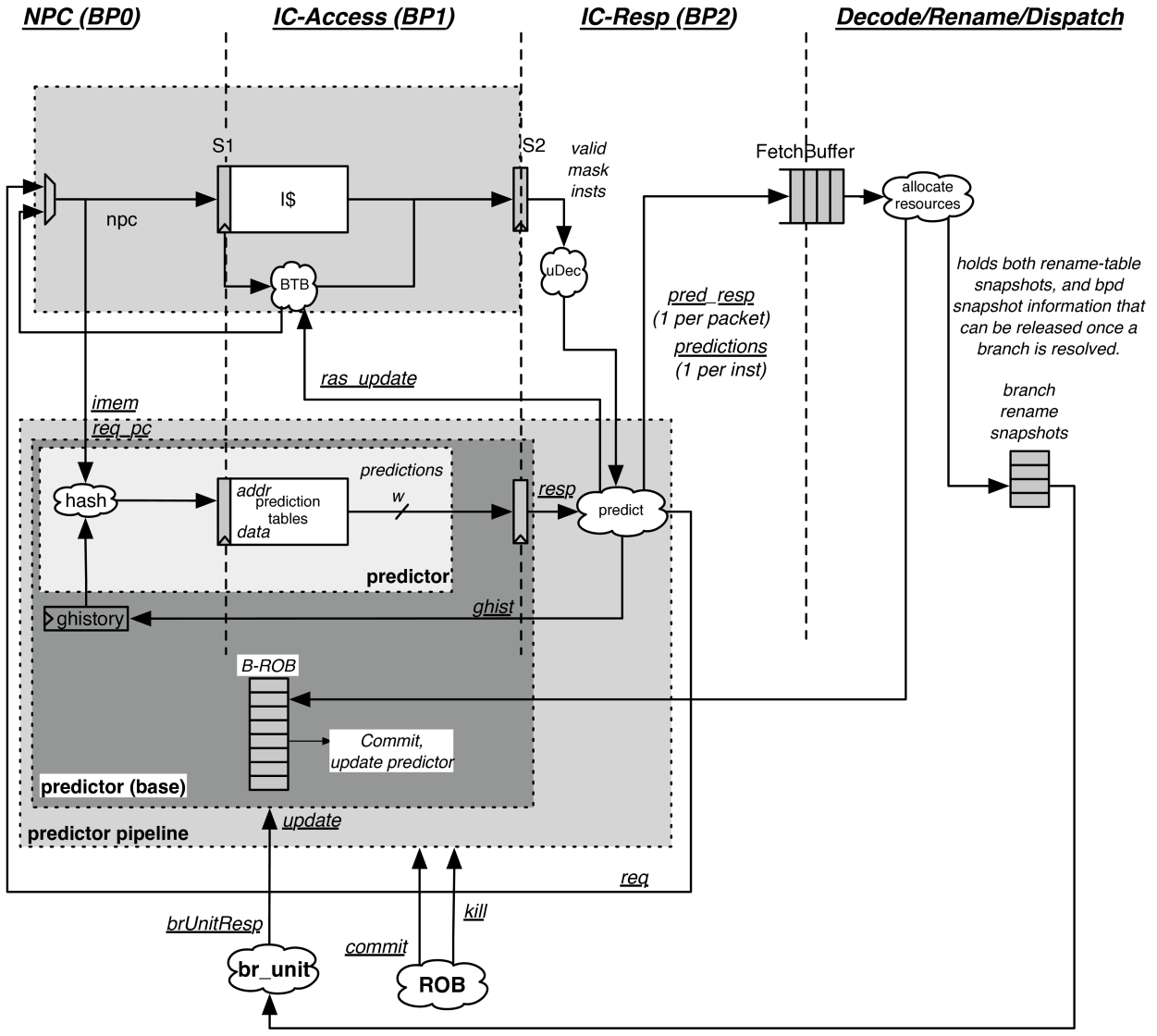


Fig. 3.3: The Backing Branch Predictor (BPD) Unit. The front-end sends the "next PC" (npc) to the BPD (BP0 stage). A hash of the npc and the global history is used to index the predictor tables. The predictor tables (ideally stored in SRAM) are accessed in parallel with the instruction cache (BP1 stage). The BPD then returns a prediction for each instruction in the fetch packet. The instructions returning from the instruction cache are quickly decoded; any branches that are predicted as taken will redirect the front- end from the BP2 stage. Prediction snapshots and metadata are stored in the branch rename snapshots (for fixing the predictor after mispredictions) and the Branch Re-order Buffer (for updating the predictor in the Commit stage).

The NLP learns any "taken" instruction's *PC* and *target PC* - thus, the NLP is able to predict jumps and jump-register instructions.

If the NLP does not make a prediction on a instruction, the pipeline will redirect the Fetch Unit in the *Fetch2 Stage* (see Fig. 2.1).[9]

Jump-register instructions that were not predicted by the NLP will be sent down the pipeline with no prediction made. As instructions require reading the register file to deduce the jump target, there's nothing that can be done if the NLP does not make a prediction.

3.2.3 Updating the Backing Predictor

Generally speaking, the BPD is updated during the *Commit* stage. This prevents the BPD from being polluted by wrong-path information.[10] However, as the BPD makes use of global history, this history must be reset whenever the Fetch Unit is redirected. Thus, the BPD must also be (partially) updated during *Execute* when a misprediction occurs to reset any speculative updates that had occurred during the *Fetch* stages.

When making a prediction, the BPD passes to the pipeline a "response info packet". This "info packet" is stored in a "branch re-order buffer" (BROB) until commit time.[11] Once all of the instructions corresponding to the "info packet" is committed, the "info packet" is set to the BPD (along with the eventual outcome of the branches) and the BPD is updated. *The Branch Reorder Buffer (BROB)* covers the BROB, which handles the snapshot information needed for update the predictor during *Commit*. *Rename Snapshot State* covers the BPD Rename Snapshots, which handles the snapshot information needed to update the predictor during a misspeculation in the *Execute* stage.

3.2.4 Managing the Global History Register

The *global history register* is an important piece of a branch predictor. It contains the outcomes of the previous branches (where is the size of the global history register).[12]

When fetching branch , it is important that the direction of the previous branches is available so an accurate prediction can be made. Waiting till the *Commit* stage to update the global history register would be too late (dozens of branches would be inflight and not reflected!). Therefore, the global history register must be updated *speculatively*, once the branch is fetched and predicted in the *BP2* stage.

If a misprediction occurs, the global history register must be reset and updated to reflect the actual history. This means that each branch (more accurately, each *fetch packet*) must snapshot the global history register in case of a misprediction.[13]

There is one final wrinkle - exceptional pipeline behavior. While each branch contains a snapshot of the global history register, any instruction can potential throw an exception that will cause a front-end redirect. Such an event will cause the global history register to become corrupted. For exceptions, this may seem acceptable - exceptions should be rare and the trap handlers will cause a pollution of the global history register anyways (from the point of view of the user code). However, some exceptional events include "pipeline replays" - events where an instruction causes a pipeline

[9] Redirecting the Fetch Unit in the *Fetch2 Stage* for instructions is trivial, as the instruction can be decoded and its target can be known.

[10] In the data-cache, it can be useful to fetch data from the wrong path- it is possible that future code executions may want to access the data. Worst case, the cache's effective capacity is reduced. But it can be quite dangerous to add wrong-path information to the BPD - it truly represents a code-path that is never exercised, so the information will *never* be useful in later code executions. Worst, aliasing is a problem in branch predictors (at most partial tag checks are used) and wrong-path information can create deconstructive aliasing problems that worsens prediction accuracy. Finally, bypassing of the inflight prediction information can occur, eliminating any penalty of not updating the predictor until the *Commit* stage.

[11] These *info packets* are not stored in the ROB for two reasons - first, they correspond to *fetch packets*, not instructions. Second, they are very expensive and so it is reasonable to size the BROB to be smaller than the ROB.

[12] Actually, the direction of all conditional branches within a *fetch packet* are compressed (via an OR-reduction) into a single bit, but for this section, it is easier to describe the history register in slightly inaccurate terms.

[13] Notice that there is a delay between beginning to make a prediction in the *BP0* stage (when the global history is read) and redirecting the front-end in the *BP2* stage (when the global history is updated). This results in a "shadow" in which a branch beginning to make a prediction in *BP0* will not see the branches (or their outcomes) that came a cycle (or two) earlier in the program (that are currently in *BP1* or *BP2* stages). It is vitally important though that these "shadow branches" be reflected in the global history snapshot.

flush and the instruction is refetched and re-executed.[14] For this reason, a *commit copy* of the global history register is also maintained by the BPD and reset on any sort of pipeline flush event.

3.2.5 The Branch Reorder Buffer (BROB)

The Reorder Buffer (see *The Reorder Buffer (ROB) and the Dispatch Stage*) maintains a record of all inflight instructions. Likewise, the Branch Reorder Buffer (BROB) maintains a record of all inflight branch predictions. These two structure are decoupled as BROB entries are *incredibly* expensive and not all ROB entries will contain a branch instruction. As only roughly one in every six instructions is a branch, the BROB can be made to have fewer entries than the ROB to leverage additional savings.

Each BROB entry corresponds to a single superscalar branch prediction. Said another way, there is a 1:1 correspondence between a single fetch cycle's prediction and a BROB entry. For each prediction made, the branch predictor packs up data that it will need later to perform an update. For example, a branch predictor will want to remember what *index* a prediction came from so it can update the counters at that index later. This data is stored in the BROB.

When the last instruction in a fetch group is committed, the BROB entry is deallocated and returned to the branch predictor. Using the data stored in the BROB entry, the branch predictor can perform any desired updates to its prediction state.

There are a number of reasons to update the branch predictor after *Commit*. It is crucial that the predictor only learns *correct* information. In a data cache, memory fetched from a wrong path execution may eventually become useful when later executions go to a different path. But for a branch predictor, wrong path updates encode information that is pure pollution – it takes up useful entries by storing information that is not useful and will never be useful. Even if later iterations do take a different path, the history that got it there will be different. And finally, while caches are fully tagged, branch predictors use partial tags (if any) and thus suffer from deconstructive aliasing.

Of course, the latency between *Fetch* and *Commit* is inconvenient and can cause extra branch mispredictions to occur if multiple loop iterations are inflight. However, the BROB could be used to bypass branch predictions to mitigate this issue. Currently, this bypass behavior is not supported in BOOM.

The BROB is broken up into two parts: the prediction *data* and the branch execution *metadata*. The metadata tracks which instructions within the fetch packet where branches, which direction they took, and which branches were mispredicted (this requires random access). The prediction data is written once into the BROB upon instruction *Dispatch* and read out (and deallocated) during *Commit*.

3.2.6 Rename Snapshot State

The BROB holds branch predictor data that will be needed to update the branch predictor during *Commit* (for both correct and incorrect predictions). However, there is additional state needed for when the branch predictor makes an incorrect prediction *and must be updated immediately*. For example, if a misprediction occurs, the speculatively-updated global history must be reset to the correct value before the processor can begin fetching (and predicting) again.

This state can be very expensive but it can be deallocated once the branch is resolved in the *Execute* stage. Therefore, the state is stored in parallel with the *Rename Snapshots*. During *Decode* and *Rename*, a branch tag is allocated to each branch and a snapshot of the rename tables are made to facilitate single-cycle rollback if a misprediction occurs. Like the branch tag and rename maptable snapshots, the corresponding branch predictor "rename" snapshot can be deallocated once the branch is resolved by the Branch Unit in *Execute*.

[14] An example of a pipeline replay is a *memory ordering failure* in which a load executed before an older store it depends on and got the wrong data. The only recovery requires flushing the entire pipeline and re-executing the load.

3.2.7 The Abstract Branch Predictor Class

To facilitate exploring different global history-based BPD designs, an abstract "BrPredictor" class is provided. It provides a standard interface into the BPD, the control logic for managing the global history register, and contains the *branch reorder buffer (BROB)* (which handles the inflight branch prediction checkpoints). This abstract class can be found in Fig. 3.3 labeled "predictor (base)".

Global History

As discussed in *Managing the Global History Register*, global history is a vital piece of any branch predictor. As such, it is handled by the abstract BranchPredictor class. Any branch predictor extending the abstract BranchPredictor class gets access to global history without having to handle snapshotting, updating, and bypassing.

Very Long Global History (VLHR)

Some branch predictors (see *The TAGE Predictor*) require access to incredibly long histories – over a thousand bits. Global history is speculatively updated after each prediction and must be snapshotted and reset if a misprediction was made. Snapshotting a thousand bits is untenable. Instead, VLHR is implemented as a circular buffer with a speculative head pointer and a commit head pointer. As a prediction is made, the prediction is written down at and the speculative head pointer is incremented and snapshotted. When a branch mispredicts, the head pointer is reset to and the correct direction is written to . In this manner, each snapshot is on the order of 10 bits, not 1000 bits.

Operating System-aware Global Histories

Although the data on its benefits are preliminary, BOOM does support OS-aware global histories. The normal global history tracks all instructions from all privilege levels. A second *user-only global history* tracks only user-level instructions.

3.2.8 The Two-bit Counter Tables

The basic building block of most branch predictors is the "Two-bit Counter Table" (2BC). As a particular branch is repeatedly taken, the counter saturates upwards to the max value 3 (*0b11*) or *strongly taken*. Likewise, repeatedly not-taken branches saturate towards zero (*0b00*). The high-order bit specifies the *prediction* and the low-order bit specifies the *hysteresis* (how "strong" the prediction is).

These two-bit counters are aggregated into a table. Ideally, a good branch predictor knows which counter to index to make the best prediction. However, to fit these two-bit counters into dense SRAM, a change is made to the 2bc finite state machine – mispredictions made in the *weakly not-taken* state move the 2bc into the *strongly taken* state (and vice versa for *weakly taken* being mispredicted). The FSM behavior is shown in Fig. 3.5.

Although it's no longer strictly a "counter", this change allows us to separate out the read and write requirements on the *prediction* and *hystersis* bits and place them in separate sequential memory tables. In hardware, the 2bc table can be implemented as follows:

The P-bit:

- **read** - every cycle to make a prediction
- **write** - only when a misprediction occurred (the value of the h-bit).

The H-bit:

- **read** - only when a misprediction occurred.
- **write** - when a branch is resolved (write the direction the branch took).

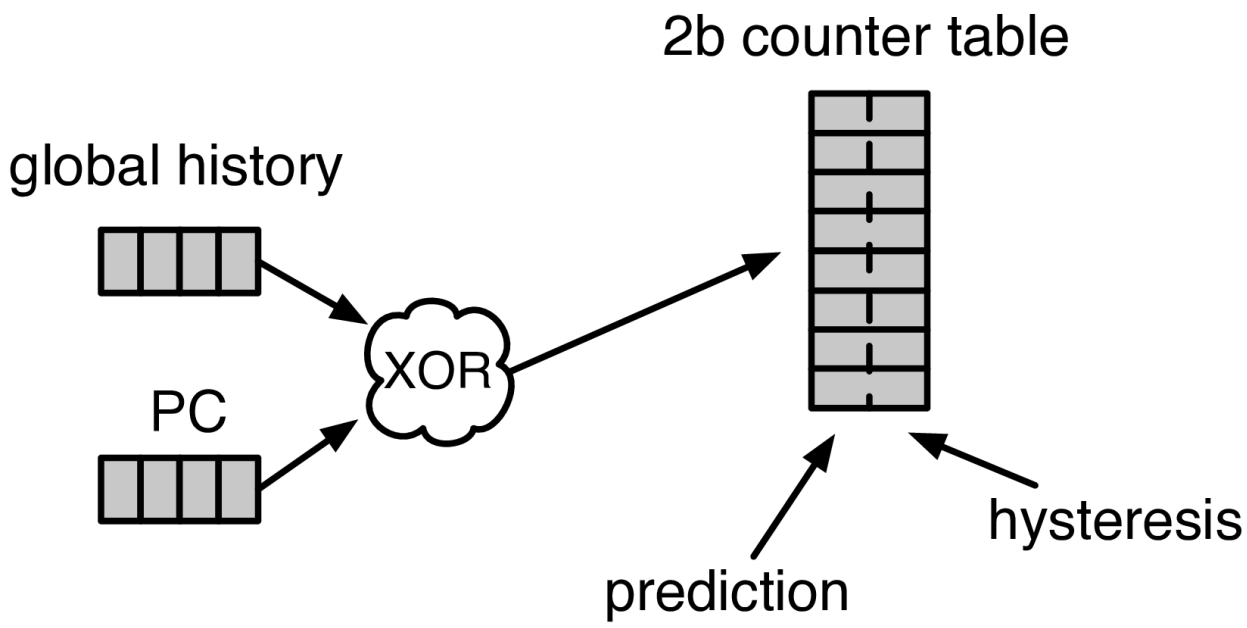


Fig. 3.4: A gshare predictor uses the global history hashed with the PC to index into a table of 2-bit counters. The high-order bit makes the prediction.

By breaking the high-order p-bit and the low-order h-bit apart, we can place each in 1 read/1 write SRAM. A few more assumptions can help us do even better. Mispredictions are rare and branch resolutions are not necessarily occurring on every cycle. Also, writes can be delayed or even dropped altogether. Therefore, the *h-table* can be implemented using a single 1rw-ported SRAM by queueing writes up and draining them when a read is not being performed. Likewise, the *p-table* can be implemented in 1rw-ported SRAM by banking it – buffer writes and drain when there is not a read conflict.

A final note: SRAMs are not happy with a "tall and skinny" aspect ratio that the 2bc tables require. However, the solution is simple – tall and skinny can be trivially transformed into a rectangular memory structure. The high-order bits of the index can correspond to the SRAM row and the low-order bits can be used to mux out the specific bits from within the row.

3.2.9 The GShare Predictor

Gshare is a simple but very effective branch predictor. Predictions are made by hashing the instruction address and the global history (typically a simple XOR) and then indexing into a table of two-bit counters. Fig. 3.4 shows the logical architecture and Fig. 3.6 shows the physical implementation and structure of the *gshare* predictor. Note that the prediction begins in the BP0 stage when the requesting address is sent to the predictor but that the prediction is made later in the BP2 stage once the instructions have returned from the instruction cache and the prediction state has been read out of the *gshare*'s p-table.

3.2.10 The TAGE Predictor

BOOM also implements the TAGE conditional branch predictor. TAGE is a highly-parameterizable, state-of-the-art global history predictor. The design is able to maintain a high degree of accuracy while scaling from very small predictor sizes to very large predictor sizes. It is fast to learn short histories while also able to learn very, very long histories (over a thousand branches of history).

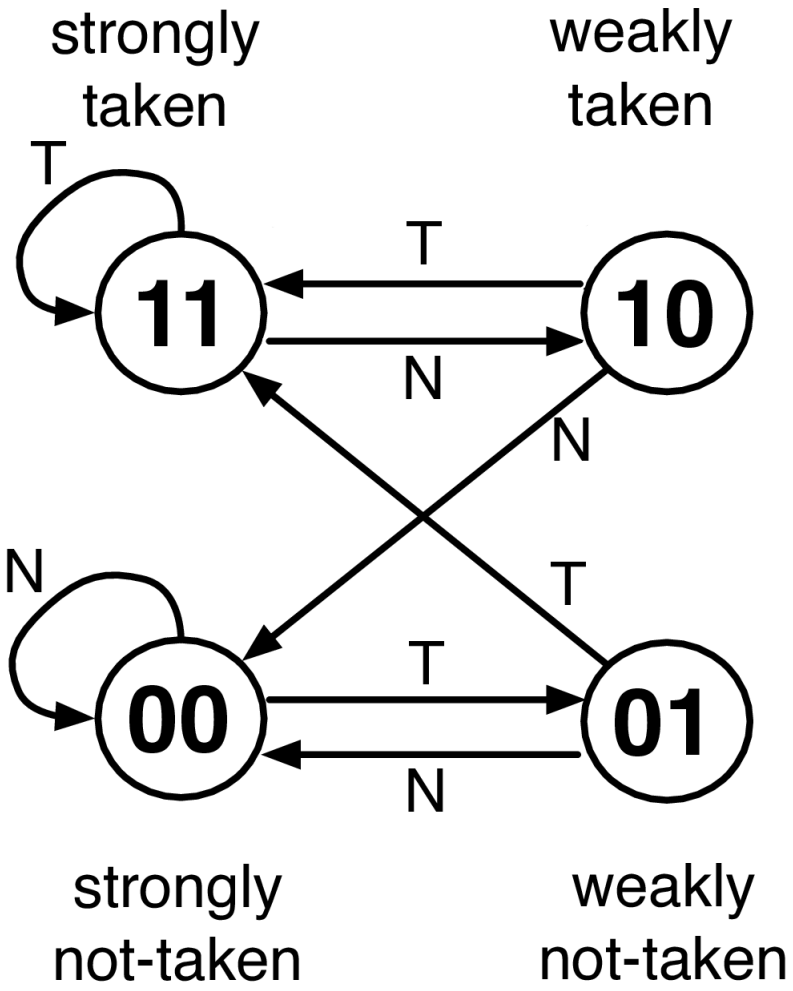


Fig. 3.5: The Two-bit counter state machine

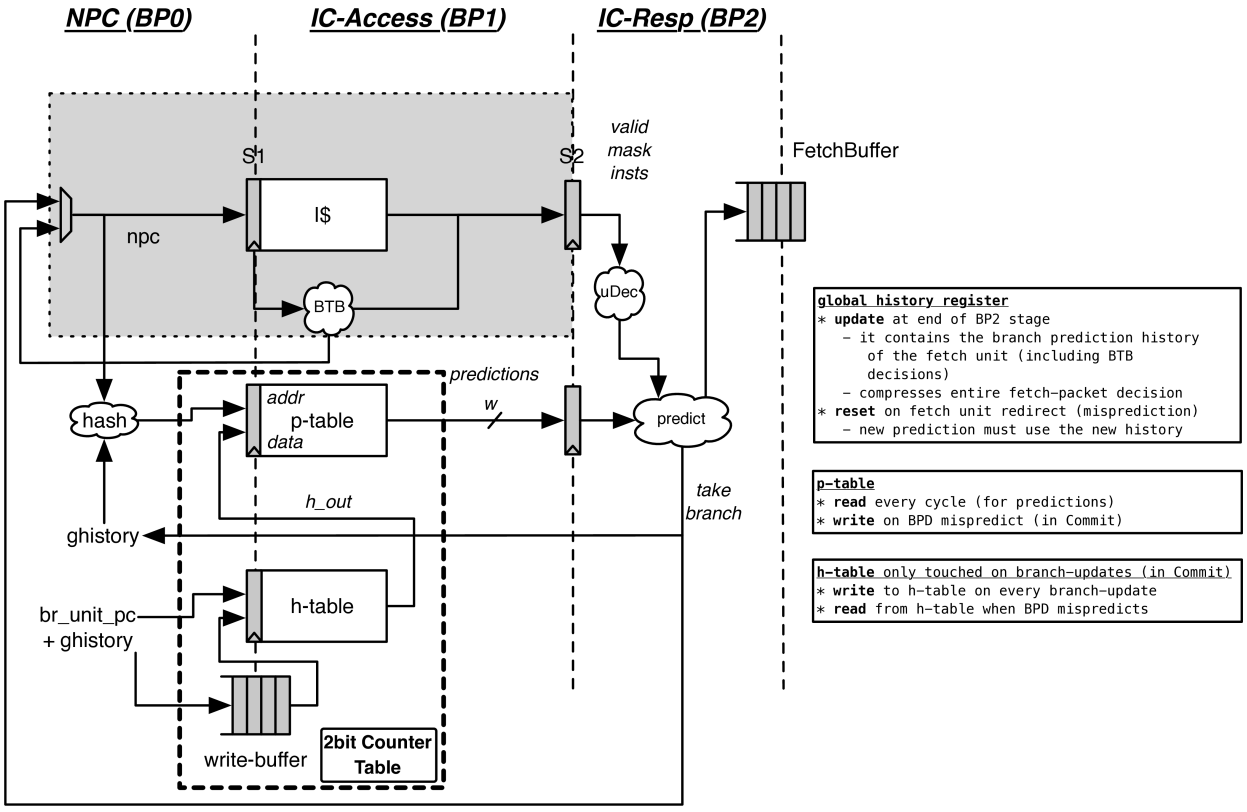


Fig. 3.6: The GShare predictor pipeline

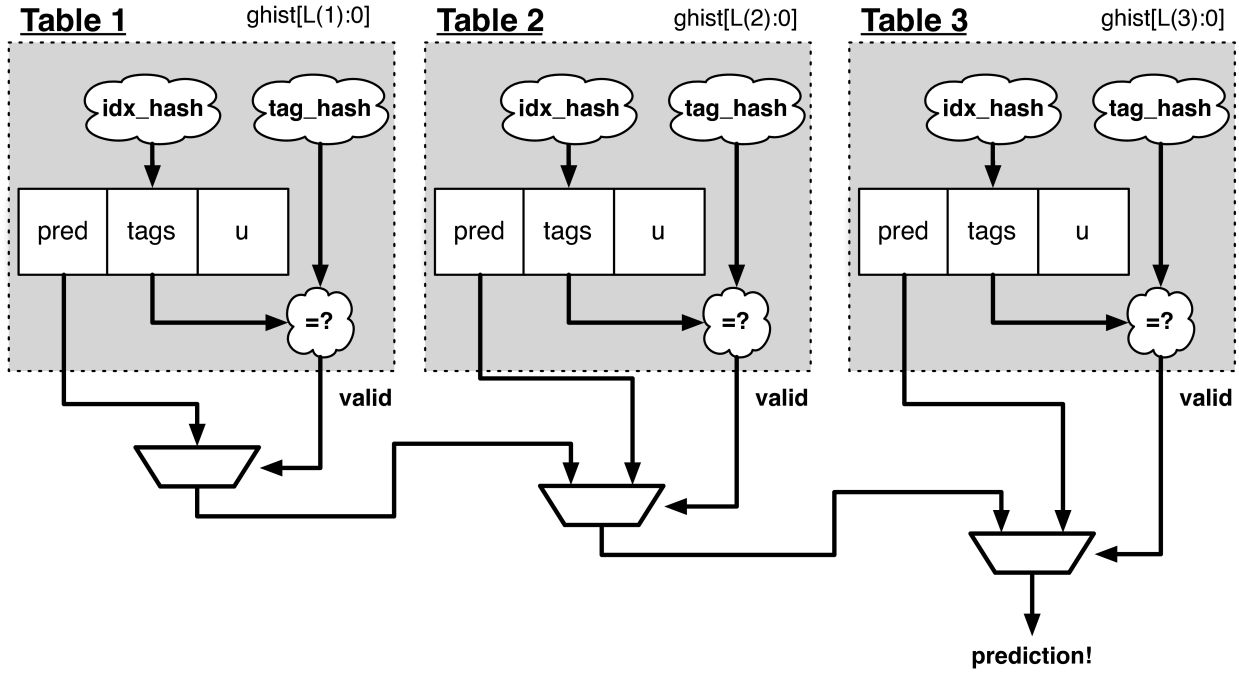


Fig. 3.7: The TAGE predictor. The requesting address (PC) and the global history are fed into each table's index hash and tag hash. Each table provides its own prediction (or no prediction) and the table with the longest history wins.

TAGE (TAgged GEometric) is implemented as a collection of predictor tables. Each table entry contains a *prediction counter*, a *usefulness counter*, and a ***tag***. The *prediction counter* provides the prediction (and maintains some hysteresis as to how strongly biased the prediction is towards taken or not-taken). The *usefulness counter* tracks how useful the particular entry has been in the past for providing correct predictions. The *tag* allows the table to only make a prediction if there is a tag match for the particular requesting instruction address and global history.

Each table has a different (and geometrically increasing) amount of history associated with it. Each table's history is used to hash with the requesting instruction address to produce an index hash and a tag hash. Each table will make its own prediction (or no prediction, if there is no tag match). The table with the longest history making a prediction wins.

On a misprediction, TAGE attempts to allocate a new entry. It will only overwrite an entry that is "not useful" ().

TAGE Global History and the Circular Shift Registers (CSRs)[15]

Each TAGE table has associated with it its own global history (and each table has geometrically more history than the last table). As the histories become incredibly long (and thus too expensive to snapshot directly), TAGE uses the Very Long Global History Register (VLHR) as described in *Very Long Global History (VLHR)*. The histories contain many more bits of history that can be used to index a TAGE table; therefore, the history must be "folded" to fit. A table with 1024 entries uses 10 bits to index the table. Therefore, if the table uses 20 bits of global history, the top 10 bits of history are XOR'ed against the bottom 10 bits of history.

Instead of attempting to dynamically fold a very long history register every cycle, the VLHR can be stored in a circular shift register (CSR). The history is stored already folded and only the new history bit and the oldest history bit need to be provided to perform an update. Listing 3.1 shows an example of how a CSR works.

Listing 3.1: The circular shift register. When a new branch outcome is added, the register is shifted (and wrapped around). The new outcome is added and the oldest bit in the history is "evicted".

```
Example:
  A 12 bit value (0b_0111_1001_1111) folded onto a 5 bit CSR becomes
  (0b_0_0010), which can be found by:


               /-- history[12] (evict bit)
               |
 c[4], c[3], c[2], c[1], c[0]
  |                        ^
  |                        |
  \_____________________/ \---history[0] (newly taken bit)


(c[4] ^ h[ 0] generates the new c[0]).
(c[1] ^ h[12] generates the new c[2]).
```

Each table must maintain *three* CSRs. The first CSR is used for computing the index hash and has a size . As a CSR contains the folded history, any periodic history pattern matching the length of the CSR will XOR to all zeroes (potentially quite common). For this reason, there are two CSRs for computing the tag hash, one of width and the other of width .

For every prediction, all three CSRs (for every table) must be snapshotted and reset if a branch misprediction occurs. Another three *commit copies* of these CSRs must be maintained to handle pipeline flushes.

[15] No relation to the Control/Status Registers.

Usefulness counters (u-bits)

The "usefulness" of an entry is stored in the *u-bit* counters. Roughly speaking, if an entry provides a correct prediction, the u-bit counter is incremented. If an entry provides an incorrect prediction, the u-bit counter is decremented. When a misprediction occurs, TAGE attempts to allocate a new entry. To prevent overwriting a useful entry, it will only allocate an entry if the existing entry has a usefulness of zero. However, if an entry allocation fails because all of the potential entries are useful, then all of the potential entries are decremented to potentially make room for an allocation in the future.

To prevent TAGE from filling up with only useful but rarely-used entries, TAGE must provide a scheme for "degrading" the u-bits over time. A number of schemes are available. One option is a timer that periodically degrades the u-bit counters. Another option is to track the number of failed allocations (incrementing on a failed allocation and decremented on a successful allocation). Once the counter has saturated, all u-bits are degraded.

TAGE Snapshot State

For every prediction, all three CSRs (for every table) must be snapshotted and reset if a branch misprediction occurs. TAGE must also remember the index of each table that was checked for a prediction (so the correct entry for each table can be updated later). Finally, TAGE must remember the tag computed for each table – the tags will be needed later if a new entry is to be allocated.[16]

3.2.11 Other Predictors

BOOM provides a number of other predictors that may provide useful.

The Null Predictor

The Null Predictor is used when no BPD predictor is desired. It will always predict "not taken".

The Random Predictor

The Random Predictor uses an LFSR to randomize both "was a prediction made?" and "which direction each branch in the *fetch packet* should take?". This is very useful for both torturing-testing BOOM and for providing a worse-case performance baseline for comparing branch predictors.

3.3 Branch Prediction Configurations

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

3.3.1 GShare Configuration Options

Global History Length

How long of a history should be tracked? The length of the global history sets the size of the branch predictor. An -bit history pairs with a entry two-bit counter table.

[16] There are ways to mitigate some of these costs, but this margin is too narrow to contain them.

3.3.2 TAGE Configurations

Number of TAGE Tables

How many TAGE tables should be used?

TAGE Table Sizes

What size should each TAGE table be?

TAGE Table History Lengths

How long should the global history be for each table? This should be a geometrically increasing value for each table.

TAGE Table Tag Sizes

What size should each tag be?

TAGE Table U-bit Size

How many bits should be used to describe the usefulness of an entry?

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 (RAW). 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.[1]

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.[2]

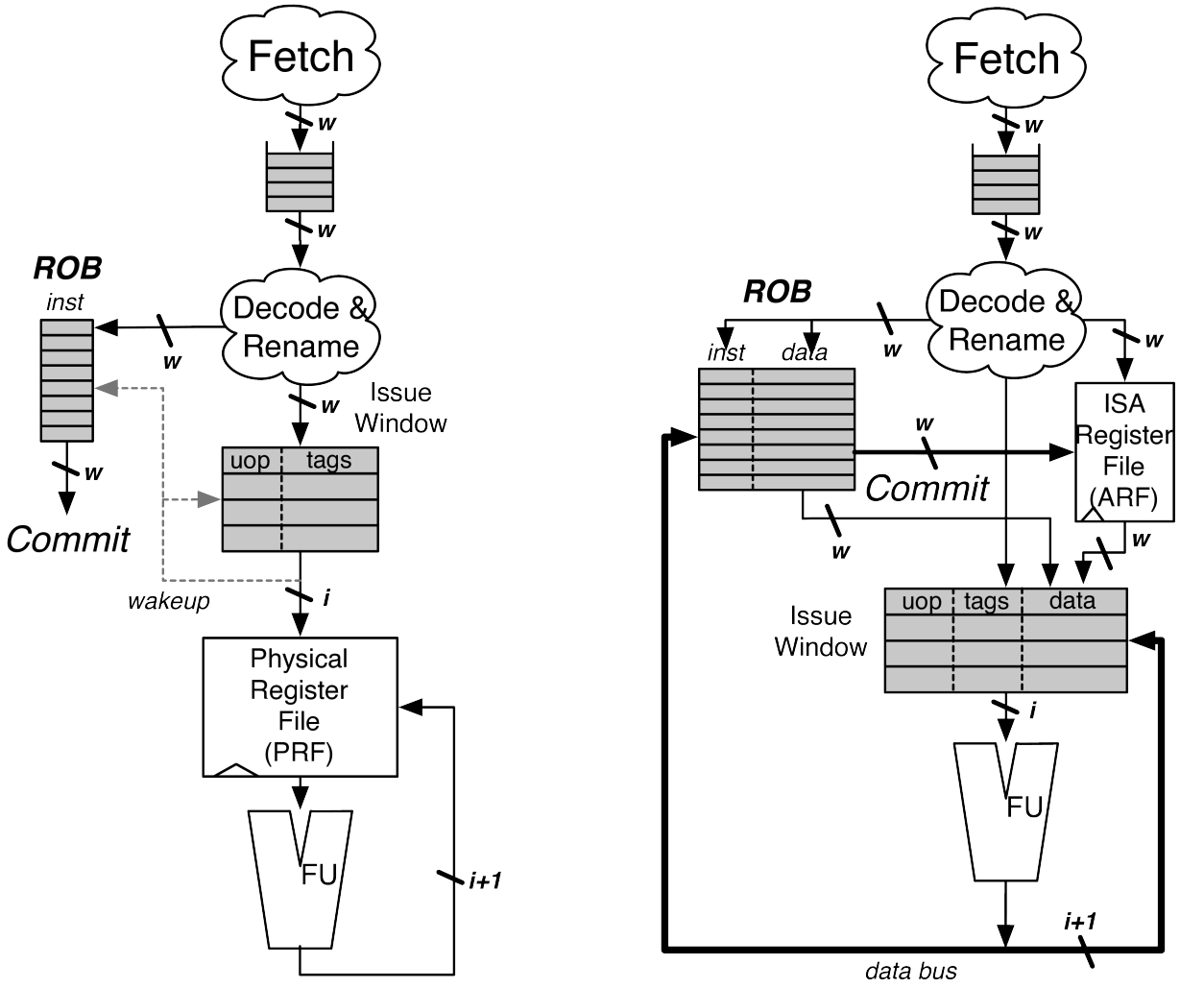


Fig. 5.1: A PRF design (left) and a data-in-ROB design (right)

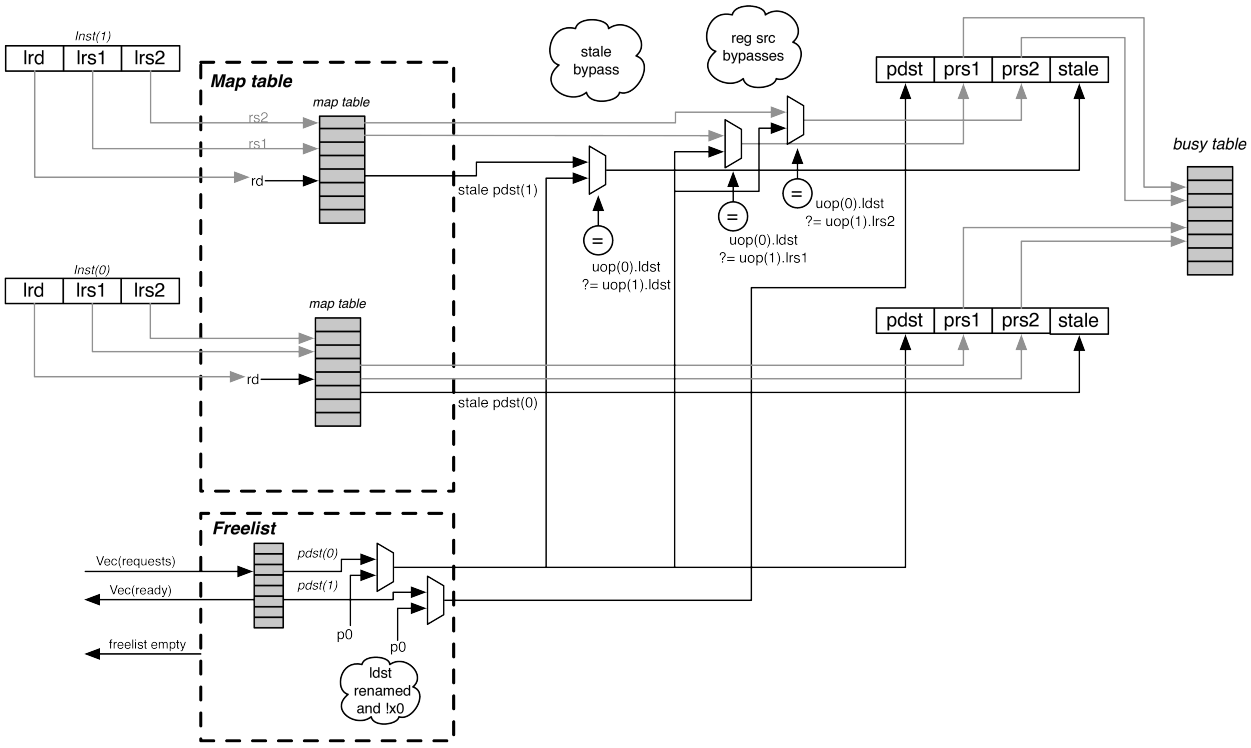


Fig. 5.2: The Rename Stage. Logical register specifiers read the map table to get their physical specifier. For superscalar rename, any changes to the map tables must be bypassed to dependent instructions. The physical source specifiers can then read the Busy Table. The Stale specifier is used to track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

5.3 The Rename Map Table

The Rename Map Table holds the speculative mappings from ISA registers to physical registers.

Each branch gets its own copy of the rename map table.[3] 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 table 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). Otherwise, pipeline flushes require multiple cycles to "unwind" the ROB to write back in the rename state at the commit point, one ROB row per cycle.

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.

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.[4]

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]

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.

[1] The MIPS R10k:raw-latex:*cite{mipsr10k}*, Alpha 21264:raw-latex:*cite{alpha21264}*, Intel Sandy Bridge, and ARM Cortex A15 cores are all example of explicit renaming out-of-order cores.

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

[3] 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.

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

[5] Conceptually, branches are often described as "snapshotting" the Free List (along with an *OR'ing* with the current Free List at the time of the misprediction). However, snapshotting fails to account for physical registers that were allocated when the snapshot occurs, then become freed, then becomes re-allocated before the branch mispredict is detected. In this scenario, the physical register gets leaked, as neither the snapshot nor the current Free List know that it had been freed. Eventually, the processor slows as it struggles to maintain enough inflight physical registers, until finally the machine comes to a halt. If this sounds autobiographical because the author may have trusted computer architecture lectures, well...

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 decoded 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 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 banks (where is the *dispatch* and *commit* width of the machine[1]). This organization is shown in Fig. 6.1.

At *dispatch*, up to 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 bubbles in the ROB, it makes adding more instructions to the ROB very cheap as the expensive costs are amortized across each ROB row.

6.2 ROB State

Each ROB entry contains relatively little state:

- is entry valid?

[1] This design sets up the *Dispatch* and *Commit* widths of BOOM to be the same. However, that is not necessarily a fundamental constraint, and it would be possible to orthogonalize the *Dispatch* and *Commit* widths, just with more added control complexity.

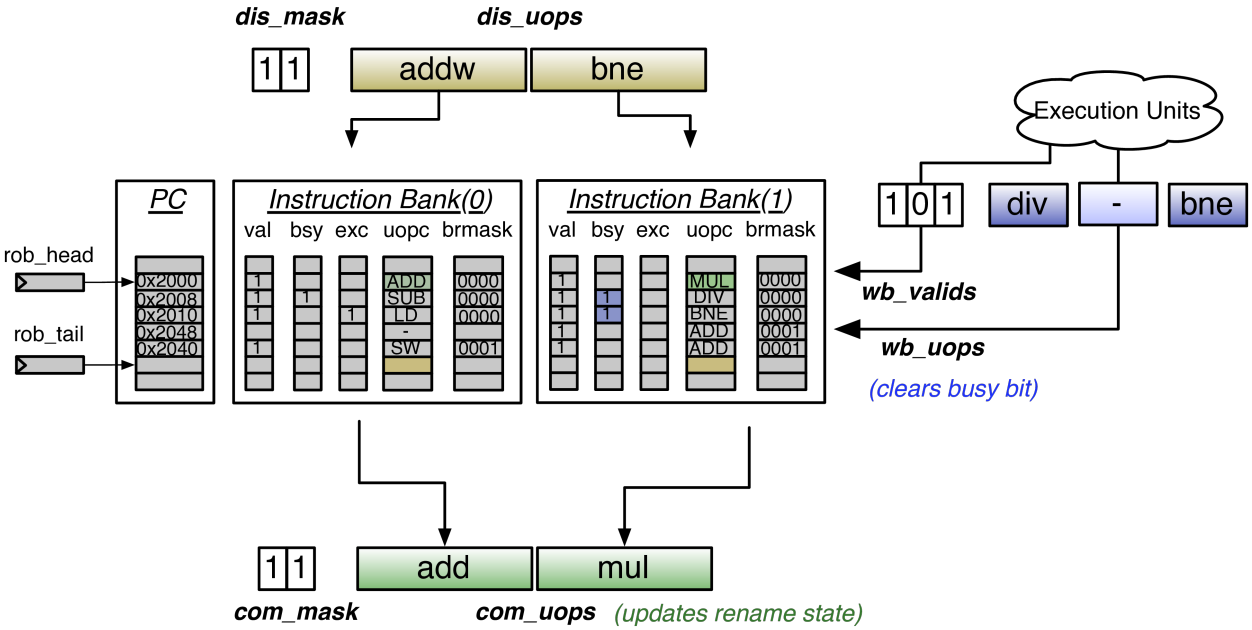


Fig. 6.1: The Reorder Buffer for a two-wide BOOM with three-issue. Dispatched uops (dis uops) are written at the bottom of the ROB (rob tail), while committed uops (com uops) are committed from the top, at rob head, and update the rename state. Uops that finish executing (wb uops) clear their busy bit. 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 busy?
- 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 *PC Storage*). The Exception State only tracks the oldest known excepting instruction (see *Exception State*).

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.[2]
- 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*, i.e., its changes to the architectural state of the machine are 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 it is an architectural exception, the excepting instruction's PC (referred to as the *exception vector*) is sent to the Control/Status Register file. If it is a micro-architectural exception (e.g., a load/store ordering misspeculation) the failing instruction is refetched and execution can begin anew.

6.4.1 Parameterization - Rollback versus Single-cycle Reset

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

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.[3]

6.4.2 Causes

The RV64G ISA provides relatively few exception sources:

Load/Store Unit

[2] Because instructions within an ROB row are consecutive in the program, the instruction's ROB bank implicitly provides the lower PC bits.
[3] The tradeoff here is between longer latencies on exceptions versus an increase in area and wiring.

- page faults

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 also originate from the Load/Store Unit, and are treated as exceptions in the BOOM pipeline (actually they 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

Fig. 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*.

[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.

7.3 Issue Select Logic

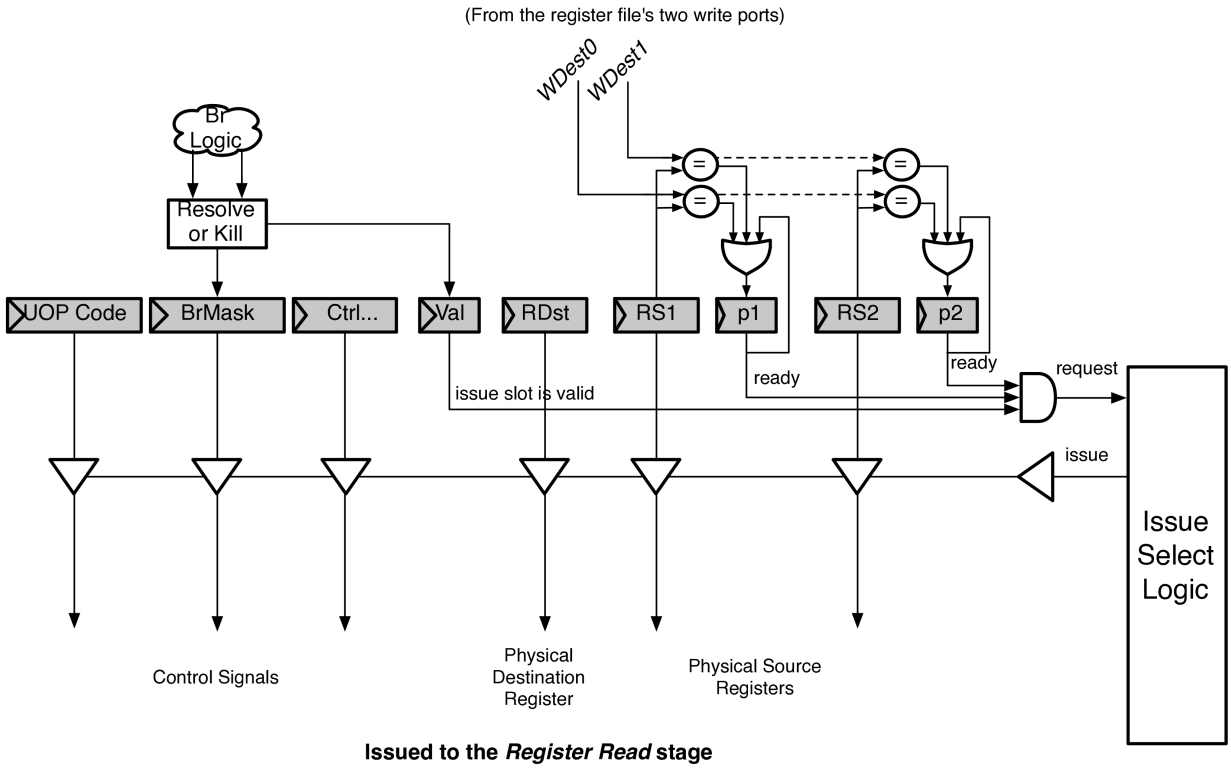


Fig. 7.1: A single issue slot from the Issue Window.

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 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. 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

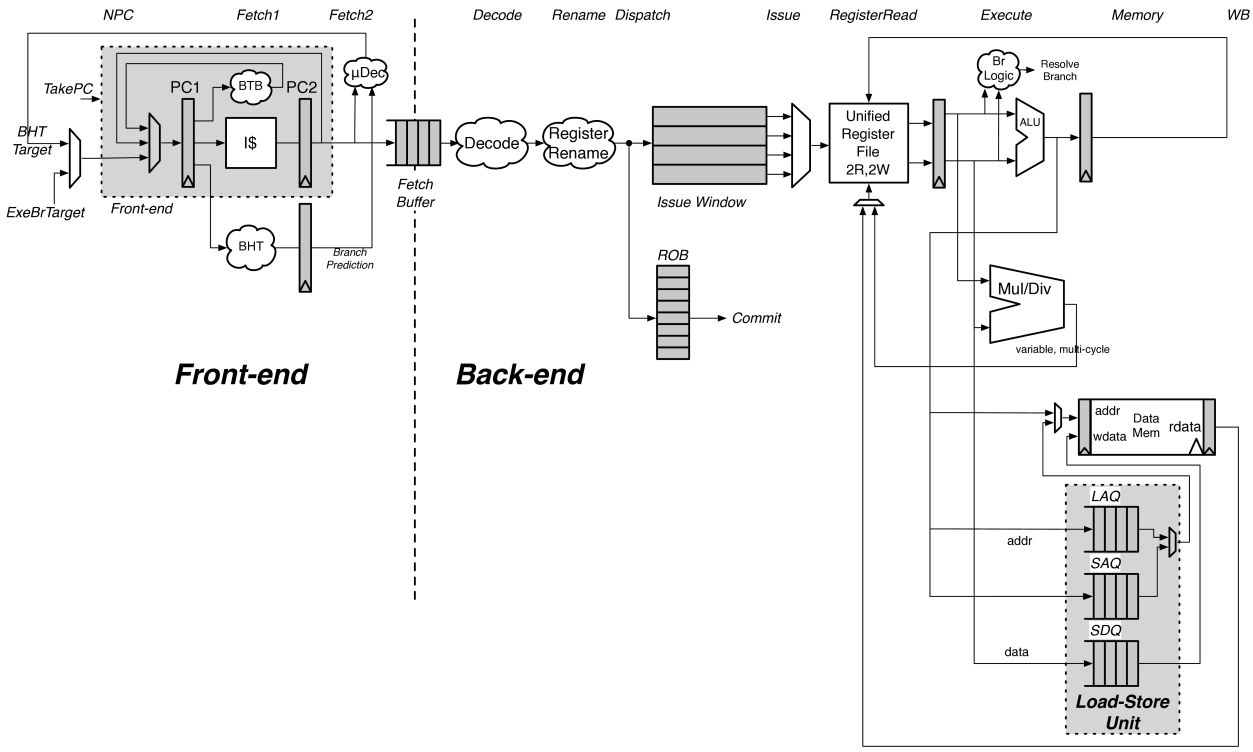


Fig. 8.1: An example single-issue pipeline. The register file needs 3 read ports to satisfy FMA operations and 2 write ports to satisfy the variable latency operations without interfering with the fixed latency ALU and FP write-backs. To make scheduling of the write port trivial, the ALU's pipeline is 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.

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.

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

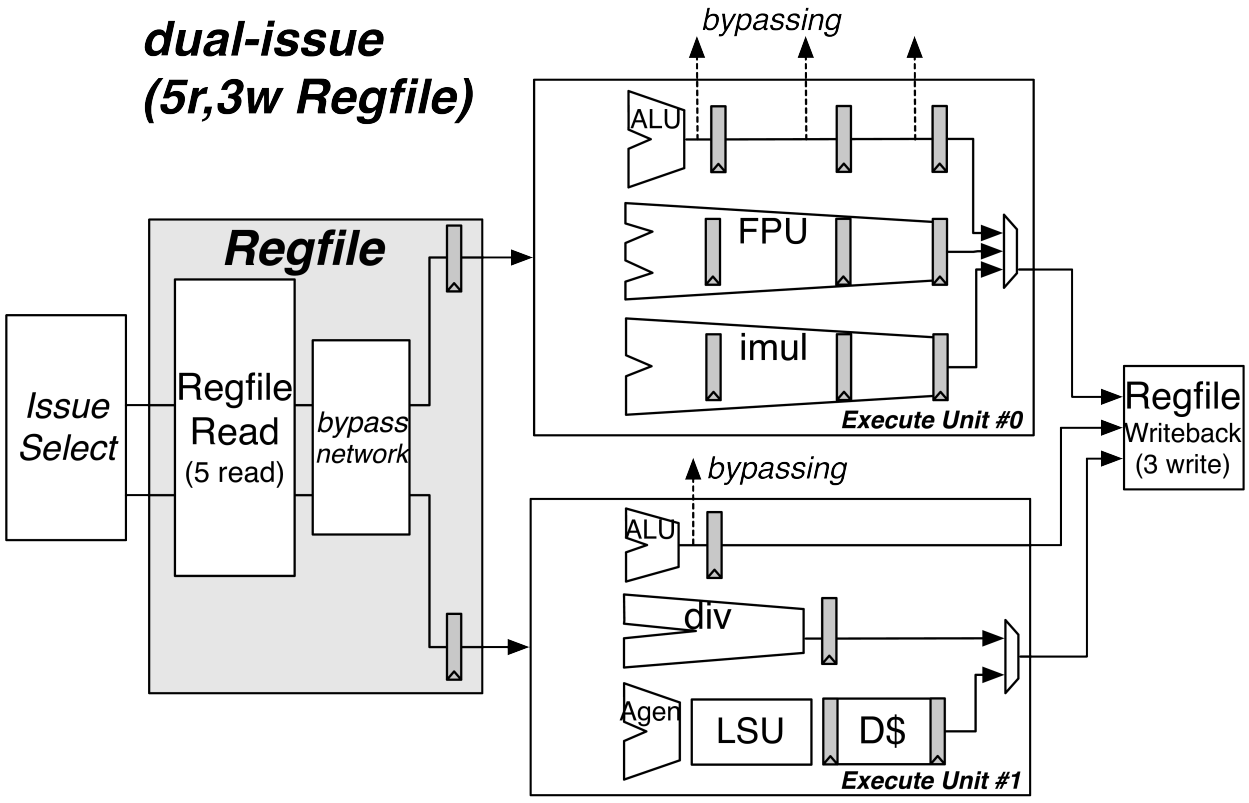


Fig. 9.1: An example pipeline for a dual-issue BOOM. The first issue port schedules micro-ops onto Execute Unit #0, which can accept ALU operations, FPU operations, and integer multiply instructions. The second issue port schedules ALU operations, integer divide instructions (unpipelined), and load/store 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.

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. Fig. 9.1 shows an example Execution Pipeline for a dual-issue BOOM.

9.1 Execution Units

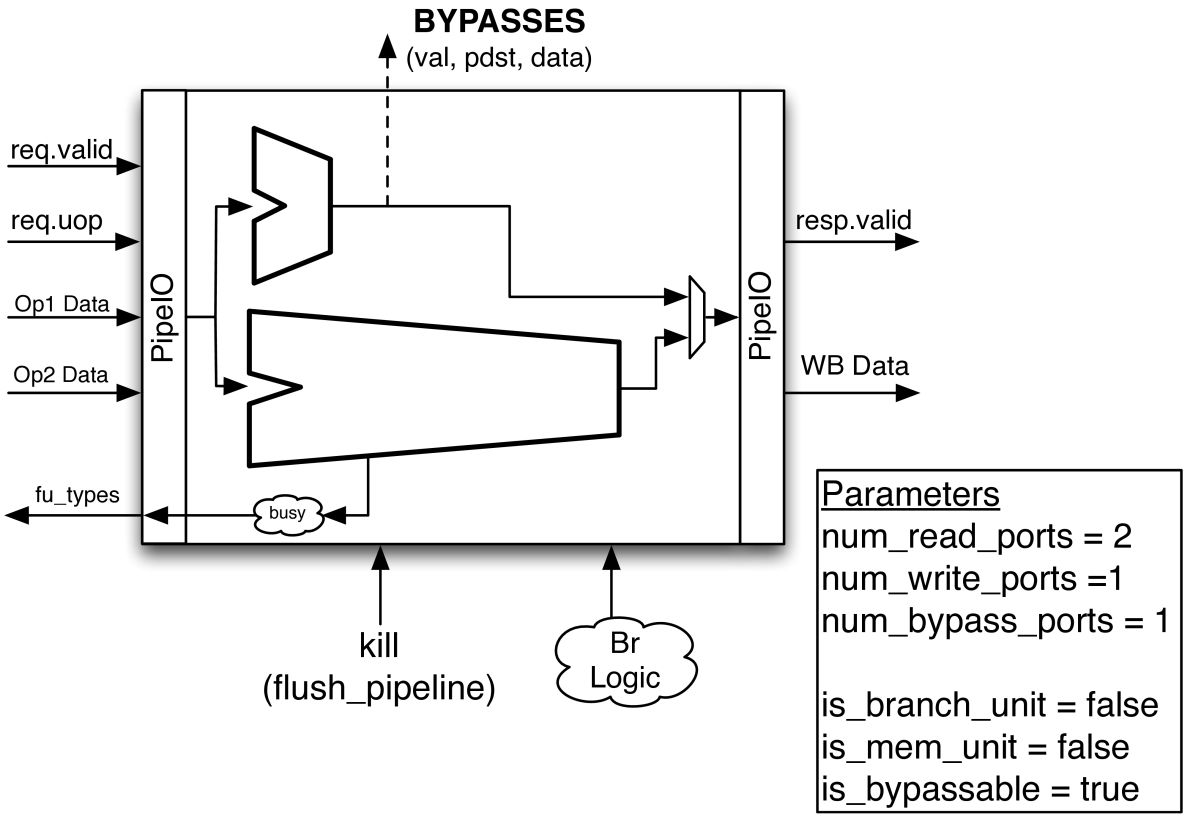


Fig. 9.2: An example Execution Unit. This particular example shows an integer ALU (that can bypass results to dependent instructions) and an unpipelined divider that becomes busy during operation. Both functional units share a single write-port. The Execution Unit accepts both kill signals and branch 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. 9.1).

9.2 Functional Units

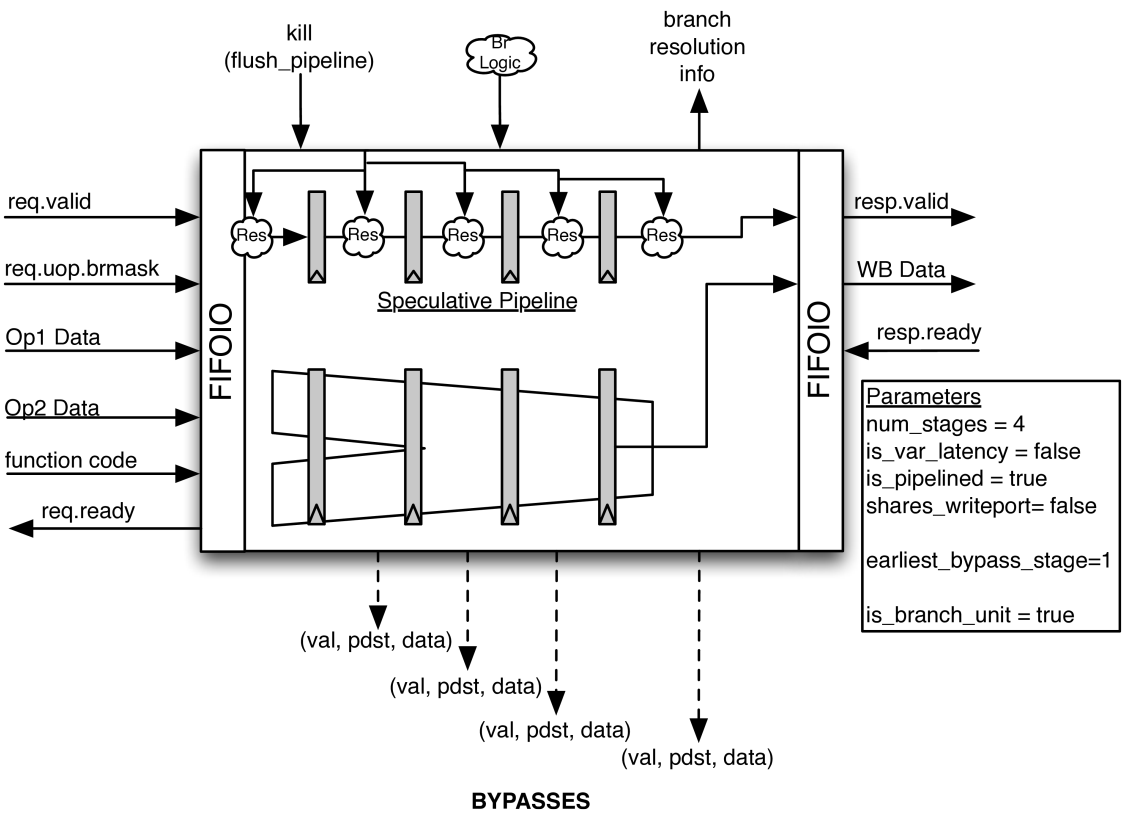


Fig. 9.3: The abstract Pipelined Functional Unit class. An expert-written, low-level functional unit is instantiated within the Functional Unit. The request and response ports are abstracted and bypass and branch speculation support is provided. Micro-ops are individually killed by gating off their response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

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.

For this reason, BOOM uses an abstract Functional Unit class to "wrap" expert-written, low-level functional units from the Rocket repository (see *The Rocket-chip Repository Layout*). 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.[1]

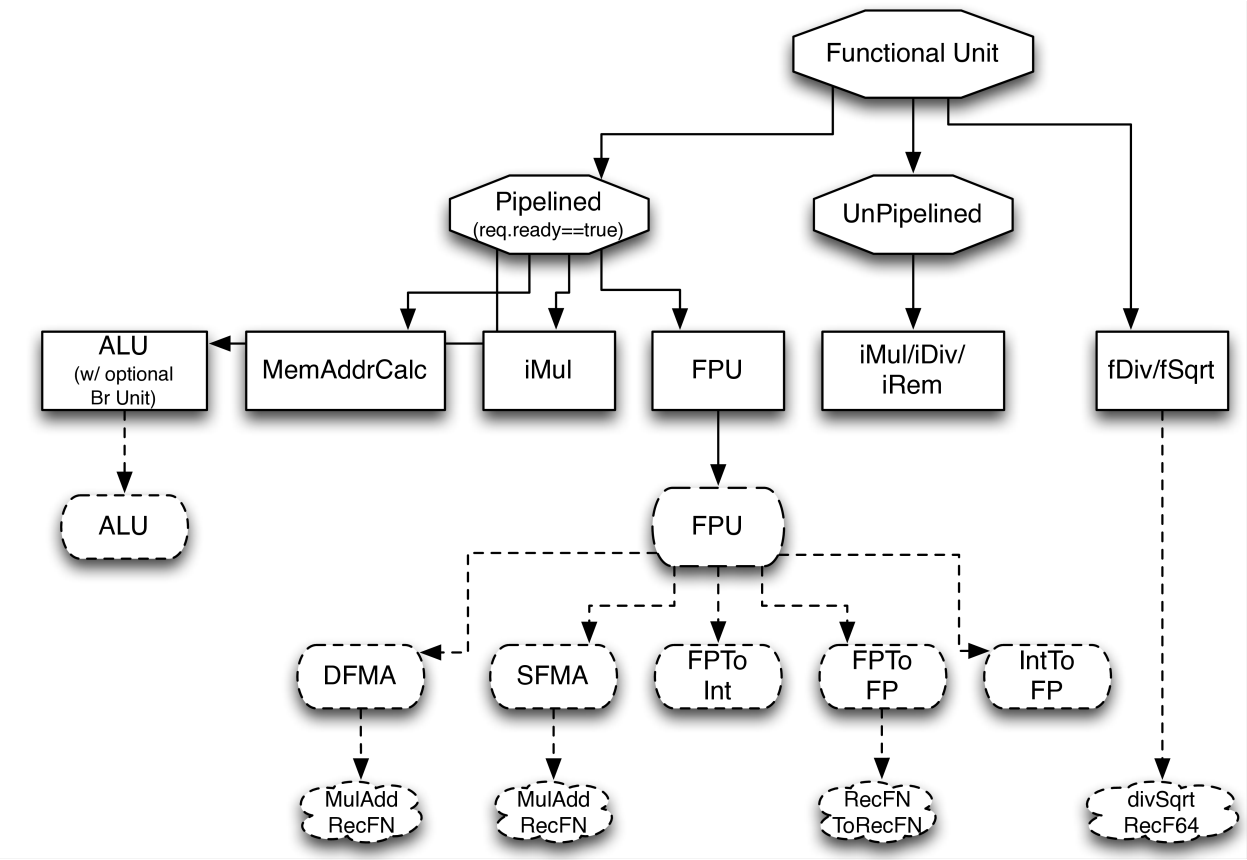


Fig. 9.4: The dashed ovals are the low-level functional units written by experts, the squares are concrete classes that instantiate the low-level functional units, and the octagons are abstract classes that provide generic speculation support and interfacing with the BOOM pipeline. The floating point divide and squart-root unit doesn't cleanly fit either the Pipelined nor Unpipelined abstract class, and so directly inherits from the FunctionalUnit super class.

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).

[1] This constraint could be relaxed by waiting for the un-pipelined unit to finish before de-asserting its busy signal and suppressing the *valid* output signal.

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).[2]

See *The Load/Store Unit (LSU)* 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/freechipsproject/rocket-chip) and hardfloat (https://github.com/ucb-bar/berkeley-hardfloat) repositories. Figure [fig:functional-unit-fpu] shows the class hierarchy of the FPU.

To make the scheduling of the write-port trivial, all of the pipelined FP units are padded to have the same latency.[3]

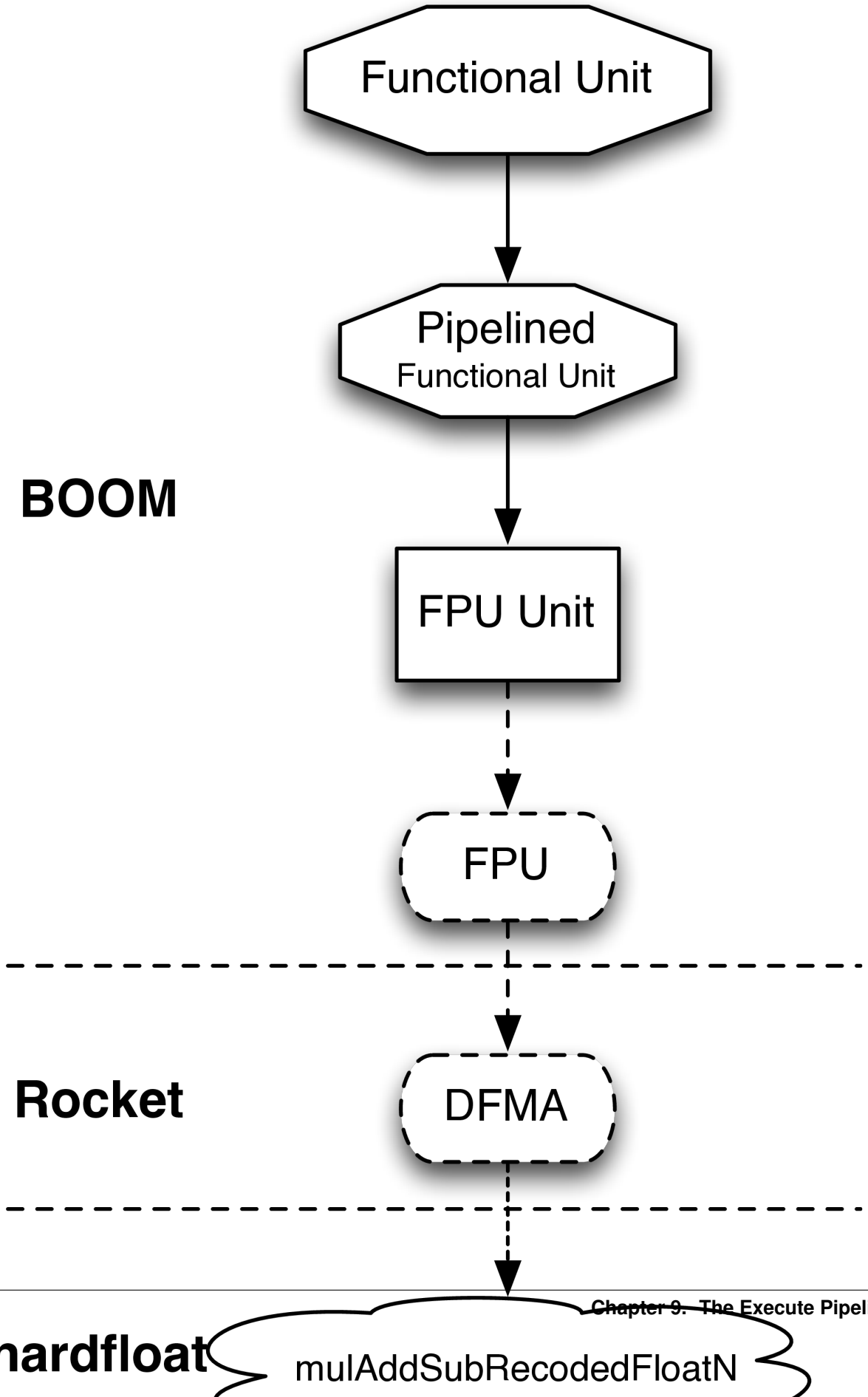
9.6 Floating Point Divide and Square-root Unit

BOOM fully supports floating point divide and square-root operations using a single **FDiv/Sqrt** (or **fdiv** for short). BOOM accomplishes this by instantiating a double-precision unit from the hardfloat repository. The unit comes with the following features/constraints:

- expects 65-bit recoded double-precision inputs
- provides a 65-bit recoded double-precision output
- can execute a divide operation and a square-root operation simultaneously
- operations are unpipelined and take an unknown, variable latency
- provides an *unstable* FIFO interface

[2] 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.

[3] 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.


Functional Unit
Pipelined
Functional Unit
BOOM
FPU Unit
FPU
Rocket
DFMA
hardfloat
mulAddSubRecodedFloatN

Single-precision operations have their operands upscaled to double-precision (and then the output downscaled).[4]

Although the unit is unpipelined, it does not fit cleanly into the Pipelined/Unpipelined abstraction used by the other functional units (see Fig. 9.4). This is because the unit provides an unstable FIFO interface: although the unit may provide a *ready* signal on Cycle , there is no guarantee that it will continue to be *ready* on Cycle , even if no operations are enqueued. This proves to be a challenge, as the issue window may attempt to issue an instruction but cannot be certain the unit will accept it once it reaches the unit on a later cycle.

The solution is to add extra buffering within the unit to hold instructions until they can be released directly into the unit. If the buffering of the unit fills up, back pressure can be safely applied to the issue window.[5]

9.7 Parameterization

BOOM provides flexibility in specifying the issue width and the mix of functional units in the execution pipeline. Code [code:exe_units] shows how to instantiate an execution pipeline in BOOM.

Listing 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.

```
val exe_units = ArrayBuffer[ExecutionUnit]()

if (ISSUE_WIDTH == 2)
{
   exe_units += Module(new ALUExeUnit(is_branch_unit = true
                                      , has_mul      = true
                                      ))
   exe_units += Module(new ALUMemExeUnit(has_div     = true
                                      ))
}
else if (ISSUE_WIDTH == 3)
{
   exe_units += Module(new ALUExeUnit(is_branch_unit = true
                                      , has_mul      = true
                                      ))
   exe_units += Module(new ALUExeUnit(has_div = true))
   exe_units += Module(new MemExeUnit())
}
```

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

9.8 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).

[4] It is cheaper to perform the SP-DP conversions than it is to instantiate a single-precision fdivSqrt unit.

[5] It is this ability to hold multiple inflight instructions within the unit simultaneously that breaks the "only one instruction at a time" assumption required by the UnpipelinedFunctionalUnit abstract class.

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.**[6] 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.[7] 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).

[6] There is a lot of room to play with regarding the CSRs. For example, it is probably a good idea to rename the 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.

[7] 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 [sec:storeuops] for more details on the store micro-op specifics.

10.1 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.

10.1.1 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.

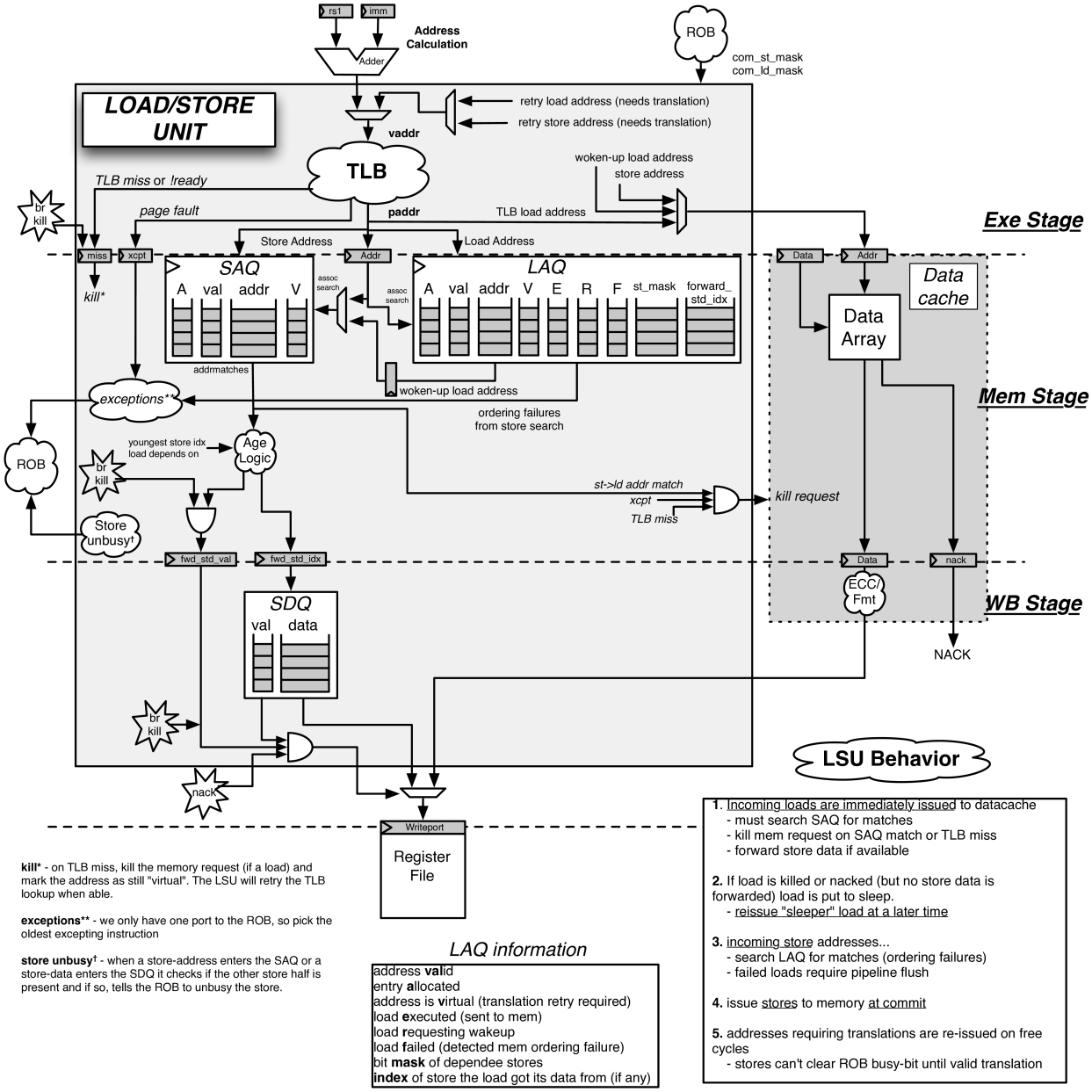


Fig. 10.1: The Load/Store Unit

10.2 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]

10.3 The BOOM Memory Model

Currently, as of October 2016, the RISC-V memory model is underspecified and will take some time to settle on an exact specification. However, the current RISC-V specification describes as *relaxed consistency* model in which stores and loads may be freely re-ordered.

BOOM currently exhibits the following behavior:

1. Write -> Read constraint is relaxed (newer loads may execute before older stores).
2. Read -> Read constraint is currently relaxed (loads to the same address may be reordered).
3. A thread can read its own writes early.

10.3.1 Ordering Loads to the Same Address

The RISC-V memory model is expected to strengthen the requirement that loads to the same address be ordered.[3] This requires loads to search against other loads for potential address conflicts. If a younger load executes before an older load with a matching address, the younger load must be replayed and the instructions after it in the pipeline flushed. However, this scenario is only required if a cache coherence probe event snooped the core's memory, exposing the reordering to the other threads. If no probe events occurred, the load re-ordering may safely occur.

10.4 Memory Ordering Failures

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

```
sw x1 0(x2)

ld x3 0(x4)
```

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

[3] Technically, a *fence.r.r* could be used to provide the correct execution of software on machines that reorder dependent loads. However, there are two reasons for an ISA to disallow re-ordering of dependent loads: 1) no other popular ISA allows this relaxation, and thus porting software to RISC-V could face extra challenges, and 2) cautious software may be too liberal with the appropriate *fence* instructions causing a slow-down in software. Thankfully, enforcing ordered dependent loads may not actually be very expensive. For one, load addresses are likely to be known early - and are probably likely to execute in-order anyways. Second, misordered loads are only a problem in the cache of a cache coherence probe, so performance penalty is likely to be negligible. The hardware cost is also negligible - loads can use the same CAM search port on the LAQ that stores must already use. While this may become an issue when supporting one load and one store address calculation per cycle, the extra CAM search port can either be mitigated via banking or will be small compared to the other hardware costs required to support more cache bandwidth.

However, if x2 and x4 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 Fig. 10.1 for more information about 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 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 (https://github.com/freechipsproject/rocket-chip).

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.

The cache has a three-stage pipeline and can accept a new request every cycle. The stages do the following:

- S0: Send request address
- S1: Access SRAM
- S2: Perform way-select and format response data

The data cache is also cache coherent which is helpful even in uniprocessor configurations for allowing a host machine or debugger to read BOOM's memory.

CHAPTER 12

Micro-architectural Event Tracking

Version 1.9.1 of the RISC-V Privileged Architecture adds support for Hardware Performance Monitor (HPM) counters.[1] The HPM support allows a nearly infinite number of micro-architectural events to be multiplexed onto up to 29 physical counters. The privilege access levels can also be set on a per-counter basis.

Table: UArch Events

Number	Event
1	Committed Branches
2	Committed Branch Mispredictions
...	and many more (see core.scala)

The available events can be modified in core.scala as desired.[2] It is then up to machine-level software to set the privilege access level and the *event selectors* for each counter as desired.

[1] Future efforts may add some counters into a memory-mapped access region. This will open up the ability to track events that, for example, may not be tied to any particular core (like last-level cache misses).

[2] Note: io.events(0) will get mapped to being Event #1 (etc.) from the point of view of the RISC-V software.

Listing 12.1: Enable Micro-arch counters

```
static void mstatus_init()
{
   // Enable user/supervisor use of perf counters
   write_csr(mucounteren, -1);
   write_csr(mscounteren, -1);

   // set events 1 and 2 to counters 3 and 4.
   write_csr(mhpmevent3, 1);
   write_csr(mhpmevent4, 2);
```

12.1 Reading HPM Counters in Software

The Code Example Listing 12.2 demonstrates how to read the value of any CSR register from software.

Listing 12.2: Read CSR Register

```
#define read_csr_safe(reg) ({ register long __tmp asm("a0"); \
        asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
        __tmp; })

long csr_cycle  = read_csr_safe(cycle);
long csr_instr  = read_csr_safe(instret);
long csr_hpmc3  = read_csr_safe(hpmcounter3);
...
long csr_hpmc31 = read_csr_safe(hpmcounter31);
```

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).

CHAPTER 14

Debugging

CHAPTER 15

Pipeline Visualization

"Pipevew" is a useful diagnostic and visualization tool for seeing how instructions are scheduled on an out-of-order pipeline.

Pipeview displays every fetched instruction and shows when it is fetched, decoded, renamed, dispatched, issued, completed, and committed ("retired"). It shows both committed and misspeculated instructions. It also shows when stores were successfully acknowledged by the memory system ("store-completion"). It is useful for programmers who wish to see how their code is performing and for architects to see which bottlenecks are constricting machine performance.

Listing 15.1: Pipeview Text Output (uncolored)

```
-(       33320000) 0x00000018a0.0 sw a0, 220(gp)
[......f.d...i...c.........r.............]-(       33320000) 0x00000018a4.0 blt s1,␣
↪a2, pc + 52
[.......f.d..i.c............r............]-(       33320000) 0x00000018a8.0 slliw a5,␣
↪a2, 2
[.......f.d...i...c.........r............]-(       33320000) 0x00000018ac.0 addw a5,␣
↪a5, a2
[........f.d...i...c.........r............]-(       33320000) 0x00000018b0.0 addiw a5,␣
↪a5, -3
[........f.d..i.c............r............]-(       33320000) 0x00000018b4.0 mv a0, a2
[.........f.d..i.c............r............]-(       33320000) 0x00000018b8.0 li a1, 3
[.........f.d...i...c.........r............]-(       33320000) 0x00000018bc.0 addi a2,␣
↪s0, -184
[..........f.di...c............r..s.......]-(       33320000) 0x00000018c0.0 sw a5, -
↪184(s0)
[..........f.d...i...c.........r.........]-(       33320000) 0x00000018c4.0 jal pc -␣
↪0x1284
[...........f.d.i.c............r........]-(       33320000) 0x0000000640.0 addiw a0,␣
↪a0, 2
[...........f.d..i.c............r........]-(       33320000) 0x0000000644.0 addw a1,␣
↪a0, a1
[............f.d.i..c............r...s...]-(       33320000) 0x0000000648.0 sw a1,␣
↪0(a2)
```

(continues on next page)

(continued from previous page)

```
[...........f.d.....i...c.....r.......]-(       33320000) 0x000000064c.0 ret
[............f.d..i..c..........r......]-(       33320000) 0x00000018c8.0 lw a2, -
↪188(s0)
[............f.d.....i...c.....r......]-(       33320000) 0x00000018cc.0 addiw a2,␣
↪a2, 1
[............f.d..i.....c........r..s..]-(       33320000) 0x00000018d0.0 sw a2, -
↪188(s0)
[............f.d.....i...c.....r.....]-(       33320000) 0x00000018d4.0 bge s1,␣
↪a2, pc - 44
[............f.d..i..c..........r....]-(       33320000) 0x00000018d8.0 lw a3, -
↪184(s0)
[............f.d...i..c.........r....]-(       33320000) 0x00000018dc.0 ld a0, -
↪200(s0)
[............f.di...c...........r...]-(       33320000) 0x00000018e0.0 addi a1,␣
↪gp, 256
[............f.d.i...c...........r...]-(       33320000) 0x00000018e4.0 jal pc -␣
↪0x1294
[...............f.d...i.c...........r..]-(       33320000) 0x0000000650.0 addiw a6,␣
↪a2, 5
[...............f.d....i...c........r..]-(       33320000) 0x0000000654.0 mv a5, a6
```

To display the text-based pipeline visualizations, BOOM generates traces compatible with the O3 Pipeline Viewer included in the gem5 simulator suite.

To enable pipeline visualization, first set O3PIPEVIEW_PRINTF in boom/src/main/scala/consts.scala to true:

```
val O3PIPEVIEW_PRINTF = true // dump trace for O3PipeView from gem5
```

Rebuild and rerun BOOM. You should find the traces (*.out) in emulator/output/. To generate the visualization, first download and install gem5, and then run:

```
boom/util/pipeview-helper.py -f <TRACE_FILE> > clean_trace.out
gem5/util/o3-pipeview.py --color --store_completions -o pipeview.out clean_trace.out
```

You can view the visualization by running:

```
less -r pipeview.out``
```

To learn more about o3-pipeview.py and to download gem5 visit http://www.m5sim.org/Visualization.

CHAPTER 16

Physical Realization

This chapter provides information useful for physically realizing the BOOM processor. Although BOOM VLSI work is very preliminary, it has been synthesized at 1 GHz on a high-end mobile 28 nm process. Unfortunately, while VLSI flows are difficult to share or make portable (and encumbered with proprietary libraries and tools), an enterprising individual may want to visit the https://github.com/ucb-bar/plsi portable "Palmer's VLSI Scripts" repository which describes one way to push BOOM through a VLSI flow.

16.1 Register Retiming

Many VLSI tools require the designer to manually specify which modules need to be analyzed for retiming.

In BOOM, the floating point units and the pipelined integer multiply unit are described combinationally and then padded to the requested latency with registers. In order to meet the desired clock frequency, **the floating point units and the pipelined integer multiply unit must be register-retimed**.

```
val mul_result = lhs.toSInt * rhs.toSInt

val mul_output_mux = MuxCase(
   UInt(0, 64), Array(
      FN(DW_64, FN_MUL)     -> mul_result(63,0),
      FN(DW_64, FN_MULH)    -> mul_result(127,64),
      FN(DW_64, FN_MULHU)   -> mul_result(127,64),
      FN(DW_64, FN_MULHSU)  -> mul_result(127,64),
      FN(DW_32, FN_MUL)     -> Cat(Fill(32, mul_result(31)), mul_result(31,0)),
      FN(DW_32, FN_MULH)    -> Cat(Fill(32, mul_result(63)), mul_result(63,32)),
      FN(DW_32, FN_MULHU)   -> Cat(Fill(32, mul_result(63)), mul_result(63,32)),
      FN(DW_32, FN_MULHSU)  -> Cat(Fill(32, mul_result(63)), mul_result(63,32))
))

io.out := ShiftRegister(mul_output_mux, imul_stages, io.valid)
```

16.2 Pipelining Configuration Options

Although BOOM does not provide high-level configurable-latency pipeline stages, BOOM does provide a few configuration options to help the implementor trade off CPI performance for cycle-time.

16.2.1 EnableFetchBufferFlowThrough

The front-end fetches instructions and places them into a *fetch buffer*. The back-end pulls instructions out of the fetch buffer and then decodes, renames, and dispatches the instructions into the *issue window*. This fetch buffer can be optionally set to be a *flow-through* queue – instructions enqueued into the buffer can be immediately dequeued on the other side on the same clock cycle. Turning this option **off** forces all instructions to spend at least one cycle in the queue but decreases the critical path between instruction fetch and dispatch.

16.2.2 EnableBrResolutionRegister

The branch unit resolves branches, detects mispredictions, fans out the branch kill signal to *all* inflight micro-ops, redirects the PC select stage to begin fetching down the correct path, and sends snapshot information to the branch predictor to reset its state properly so it can begin predicting down the correct path. Turning this option **on** delays the branch resolution by a cycle. In particular, this adds a cycle to the branch misprediction penalty (which is hopefully a rare event).

16.2.3 Functional Unit Latencies

The latencies of the pipelined floating point units and the pipelined integer multiplier unit can be modified. Currently, all floating point unit latencies are set to the latency of the longest floating point unit (i.e., the DFMA unit). This can be changed by setting the *dfmaLatency* in the *FPUConfig* class. Likewise, the integer multiplier is also set to the *dfmaLatency*.[1]

[1] The reason for this is that the imul unit is most likely sharing a write port with the DFMA unit and so must be padded out to the same length. However, this isn't fundamental and there's no reason an imul unit not sharing a write port with the FPUs should be constrained to their latencies.

CHAPTER 17

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.

17.1 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:raw-latex:*cite{hwacha}*).

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.

17.1.1 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).

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.

17.1.2 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.

17.1.3 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.

17.1.4 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).

17.2 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.

Perhaps unexpectedly, a simple but very efficient implementation could be very small. The smallest possible vector register file (four 64-bit elements per vector) weighs in at 1024 bytes. A reasonable out-of-order implementation could support 8 elements per vector and 16 inflight vector registers (for a total of 48 physical vector registers) which would only be 3 kilobytes. Following the temporal vector design of the Cray I, the vector unit can re-use the expensive scalar functional units by trading off space for time. This also opens up the vector register file to being implemented using 1 read/1 write ports, fitting it in very area-efficient SRAMs. As a point of comparison, one of the most expensive parts of a synthesizable BOOM is its flip-flop based scalar register file. While a 128-register scalar register file comes in at

1024 bytes, it must be highly ported to fully exploit scalar instruction-level parallelism (a three-issue BOOM with one FMA unit is 7 read ports and 3 write ports).

17.3 The Compressed ("C") ISA Extension

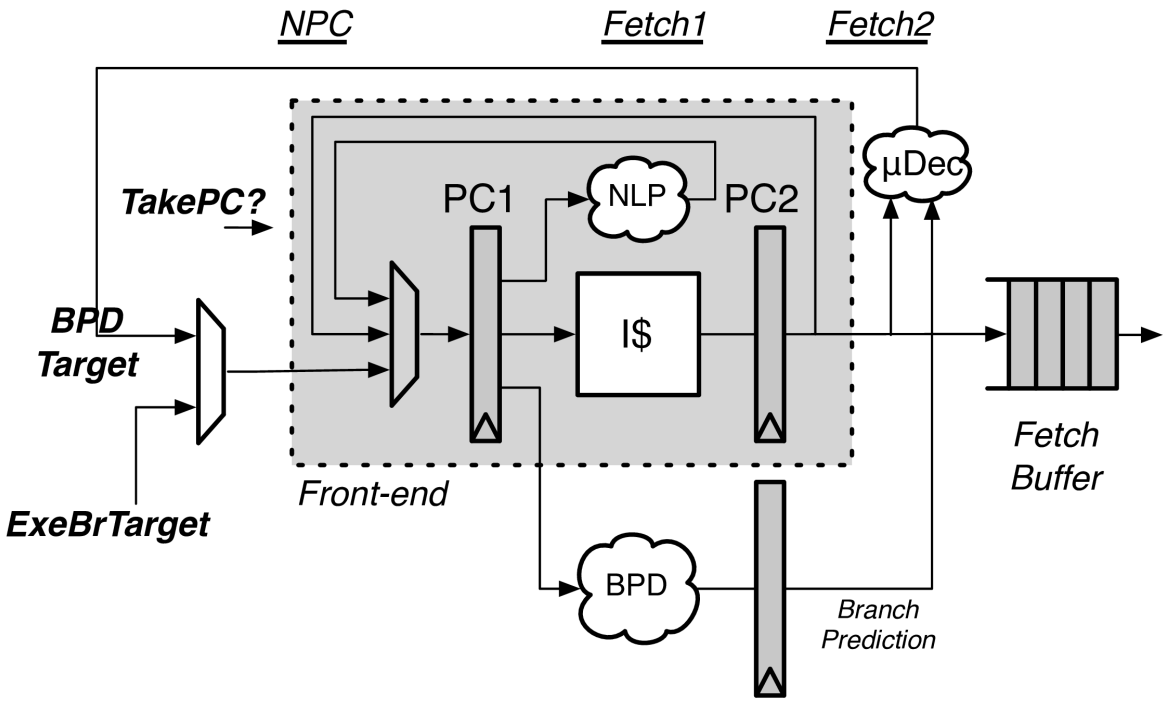


Fig. 17.1: The Fetch Unit. The grey box encompasses the Rocket front-end, which is re-used by BOOM.

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.

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 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 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 16b32b expanders at every half-word start.

17.3.1 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 or depends on whether it's the 16b JAL or a 32b JAL! Likewise, a mispredicted not-taken branch redirects the fetch unit to or 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.

CHAPTER 18

Parameterization

The width of the Decode Stage is parameterizable. However, the current limitation is the Fetch

18.1 Rocket Parameters

18.2 BOOM Parameters

18.3 Uncore Parameters

CHAPTER 19

Frequently Asked Questions

CHAPTER 20

Terminology

- 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. The {em Fetch PC} will point to the first valid instruction in the {em fetch packet}, as it is the PC used by the Front End to fetch the {em fetch packet}.
- fetch PC - The PC corresponding to the head of a {em fetch packet} instruction group.
- PC - A.k.a, the Program Counter. A weird, terrible, anachronistic term to describe the address in memory of an instruction.

CHAPTER 21

Indices and tables

- genindex
- modindex
- search