Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L15: RISC-V: Fife pipelined CPU: principles

Reminders

Please git clone or git pull: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design

```
./Book_BLang_RISCV.pdf
  Slides/
      Slides_01_Intro.pdf
      Slides_02_ISA.pdf
      ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
  Exercises/
      Ex_03_B_Top_and_DUT/
      Ex_03_A_Hello_World/
      ...
  Code/
      src_Common/
      src_Drum/
      src_Fife/
      src_Top/
      ...
```

To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- *bsc* compiler (see https://github.com/B-Lang-org/bsc)
- Verilator compiler (see https://www.verilator.org/)

Chapter Roadmap

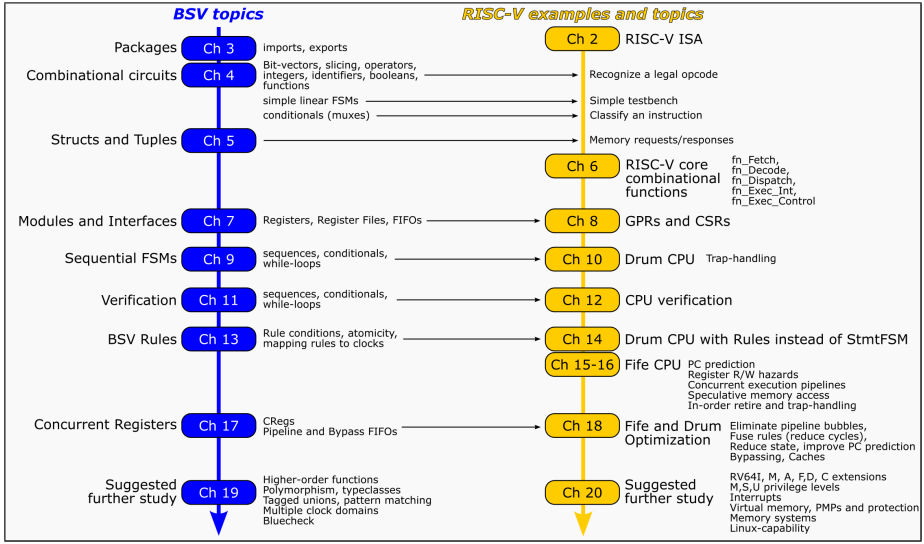
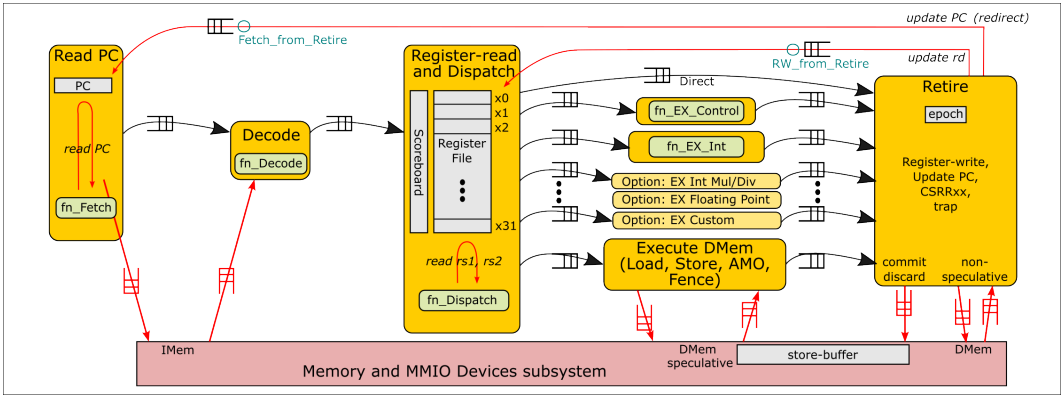


Table of Contents

Fife Instruction Flow

Rules: the fundamental behavioral construct in **BSV**



For Fife we interpret each yellow stage as an independent, infinite, concurrent process. Each stage repeatedly consumes messages from its input FIFOs, computes something, and produces messages into its output FIFOs.

Thus, multiple instructions may be "in flight", each at a different stage of its processing as it flows down the pipe.

Rules: the fundamental behavioral construct in **BSV**

Pipelining raises four new problems:

- Continuing to Fetch, with PC Prediction and Epochs
- Managing Register Read/Write Hazards with a Scoreboard
- Retiring outputs of the Execute Stages in Order, with Tags
- Allowing Memory Ops to be Pipelined with a Store Buffer

Fife: Continuing to Fetch, with PC Prediction and Epochs

Why do we need PC prediction? What should we predict?

The Fetch stage knows only the current PC, using which it can issue a read request to memory (IMem) to fetch an instruction. That instruction can have several possible outcomes:

(T) It may result in an exception (or be preempted by an external interrupt), in which the next PC is taken from CSR `mtvec`.

(B) It may be a conditional BRANCH instruction; if the branch is taken, the next PC is the branch target, otherwise it is PC+4.

(J) It may be a JAL/JALR instruction, in which case the next PC is the jump target.

(F) (In the most common case) None of the above, in which case the next PC is PC+4 ("fall-through").

But these outcomes are known only after several cycles, as the instruction flows down the pipe.

What should the Fetch stage do, in the meantime? To continue fetching, it must *predict* (guess) the next PC.

Since (F) is the most common case, PC+4 is a reasonably good prediction.

What happens when we mispredict?

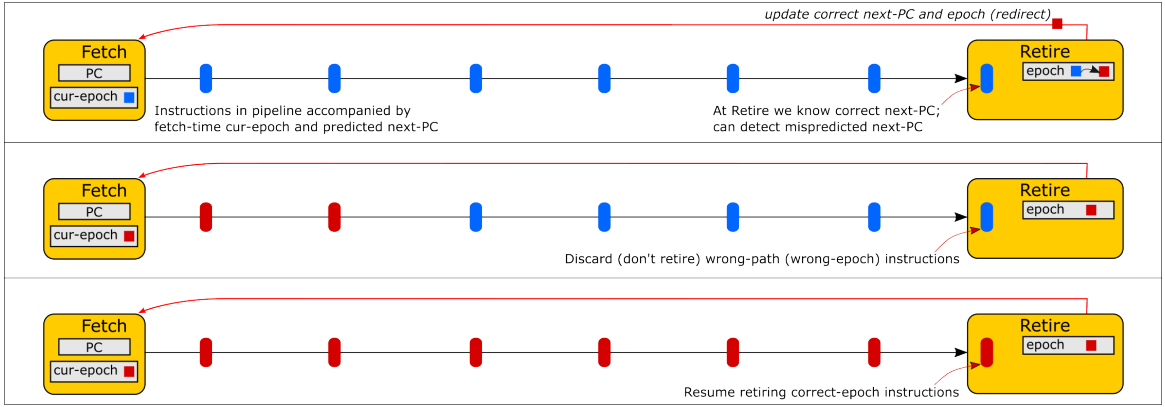
Suppose we mispredict (predict wrong) for instruction *n*.

Then,

- When we know the correct next-PC for *n*, the Fetch stage must be "*redirected*" to the correct next-PC from which it can resume fetching.
- Instructions fetched predictively after *n* (instructions at PC+4, PC+8, ...) and prior to redirection are "*wrong path*" instructions, and must be discarded.

  Equally important: the wrong-paths instructions *should not modify architectural state* (registers, CSRs, memory).

Using “epochs” to manage mispredictions



Fife: Managing Register Read/Write Hazards with a Scoreboard

The problem: Register Read/Write Hazards

- Consider an instruction I in the Register-Read-and-Dispatch stage that needs to read register x13 (Rs1 or Rs2).
- The prior instruction that writes to x13 may still be in the pipeline, ahead of I, in the Execute or Retire stage, and may not yet have written to x13.
- Instruction I must *wait* until x13 has been written, otherwise it will read a wrong (old) value.

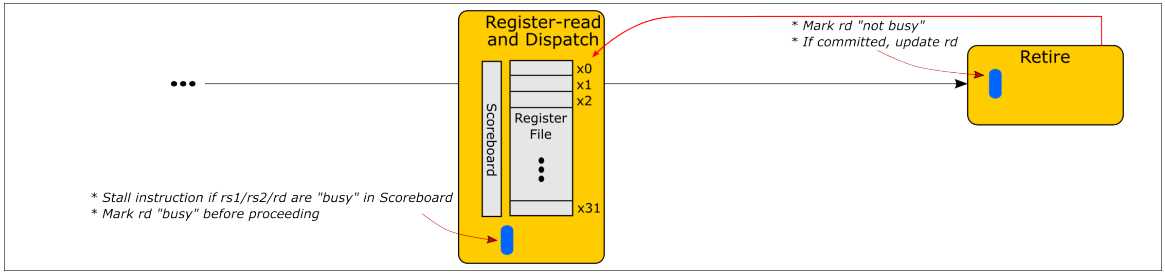
This is called a *hazard* or "race condition".

ISA semantics specify that an instruction reads and writes the architectural state *atomically*.

But a pipelined implementation may *interleave* the reads and writes of two instructions unless we take care.

One solution is to use a *scoreboard*.

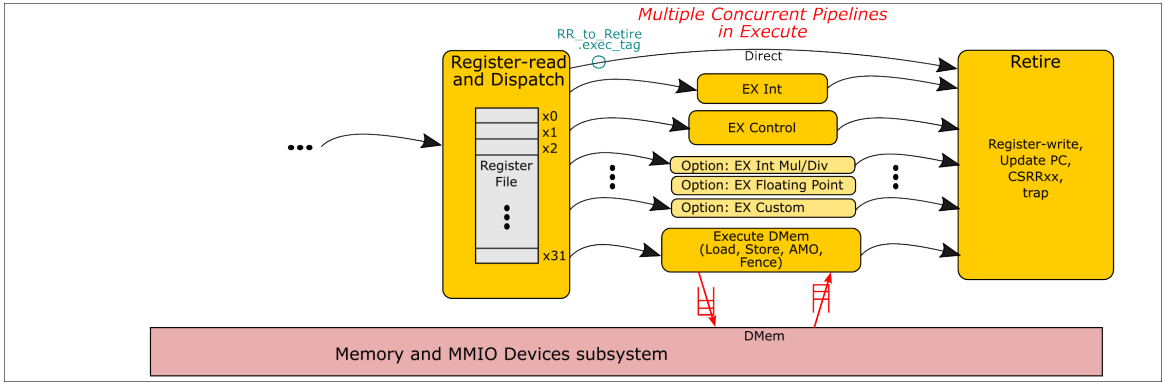
Managing Register Read/Write Hazards with a Scoreboard



Note: an instruction's Rd must be marked "not busy" whether the instruction is committed or discarded.

Fife: Retiring outputs of the Execute Stages in Order, with Tags

The problem: varying latencies in Execute pipes



Instructions may not arrive at Retire in the same order that they were Dispatched.

(In Fife, each pipe is FIFO; but the different pipes may have different latencies.)

Solution: Tags on Direct path

- In Dispatch, even if an instruction is sent into one of Execute pipes, we *also always* send information on the Direct path (`RR_to_Retire` struct).
- The `RR_to_Retire` struct contains an `exec_tag` field that indicates which Execute pipe (if any) is being used for this instruction.
- The Retire stage observes the `RR_to_Retire` struct from Direct in order to determine which Execution pipe's output (if any) contains the output of the instruction.

  The Retire stage simultaneously dequeues both the FIFO from Direct as well as the corresponding FIFO from an Execute pipe.

Thus, if an instruction through an Execute pipe arrives "early" at Retire, it will simply wait there in its FIFO until it is dequeued by Retire in the correct order.

All instructions in the pipelines are considered "speculative" until they are Retired (they may be wrong-path instructions; an instruction in front of them may trap; or the CPU may take an interrupt).

Side-effects of speculative instructions (register writes, memory writes) must never be made permanent until they are Retired.

Fife: Allowing Memory Ops to be Pipelined, with a Store Buffer

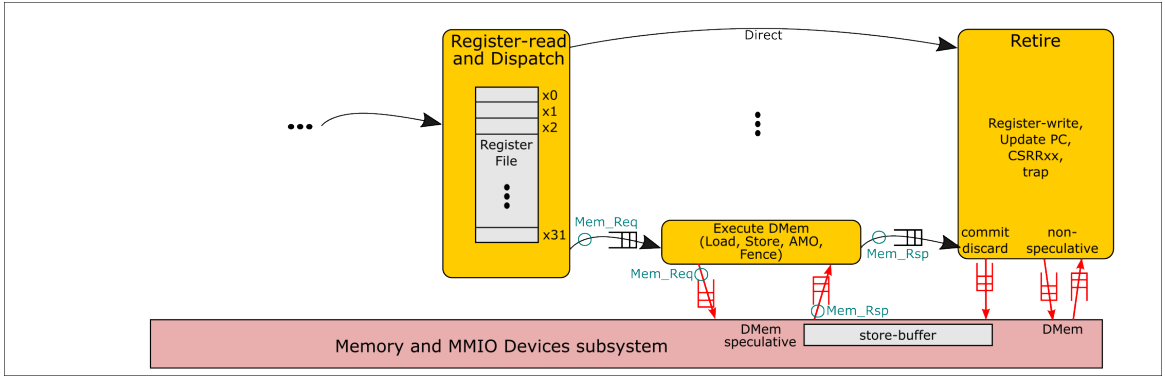
The problem: Speculative memory-writes; speculative MMIO reads and writes

- Similar to register read-write hazards, we can have memory read-write hazards—a STORE instruction may be closely followed by a LOAD/STORE instruction for the same (or overlapping) memory address.

- MMIO reads and writes may not be " memory-like" at all:
  - MMIO reads can have side effects.
  - After an MMIO write, an MMIO read from that address may not return the most-recently–written value.
  - Two MMIO reads to the same address may not return the same value (so, cannot be cached).
  - Two MMIO writes of the same value to the same address cannot be coalesced into one.

  As a result, it is too difficult to perform an MMIO LOAD/STORE speculatively; we perform it later, at Retire, when it is no longer speculative.

Solution: Store Buffer (for non-MMIO accesses)



The *Store Buffer* is a queue of pending non-MMIO STOREs to memory.

Solution: Store Buffer (for non-MMIO accesses)

The *Store Buffer* is a queue of pending non-MMIO STOREs to memory.

In Execute DMem (where everything is still speculative):

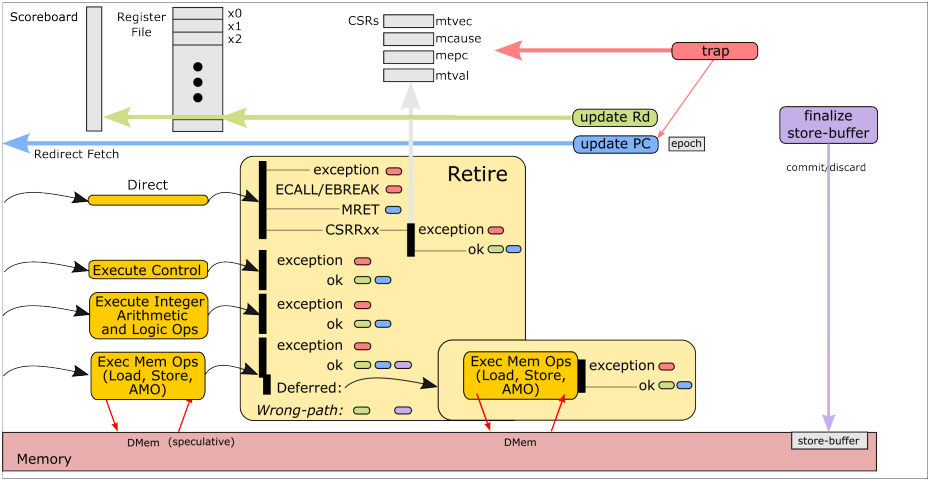
- A memory (non-MMIO) LOAD first searches the Store Buffer (from newest to oldest) in case there were recent STOREs to an overlapping address. For non-overlapping addresses, it accesses the memory system as usual.
- A memory (non-MMIO) STORE enqueues the stored value into the tail of the Store Buffer.
- An MMIO LOAD/STORE is *deferred* (returned as-is) in the `Mem_Rsp`. It will be handled later in Retire.

In Retire:

- For a discarded memory (non-MMIO) store, we send a "discard" message to the Store Bufffer; the first item in the Store Buffer can be discarded.
- For a memory (non-MMIO) STORE, we send a "commit" message to the Store Buffer; the first item in the Store Buffer can be sent into the memory system.
- For a discarded MMIO store, we just discard it.
- For a deferred MMIO LOAD/STORE, we perform the memory action (and handle an ensuing trap, if any).

Fife: Retire stage

Fife: Retire stage



Similar to Drum, with some differences for prediction/misprediction and speculative/non-speculative DMem (next slide).

Fife: Retire stage

Differences from Drum:

- "update PC" sends a redirect message to Fetch *only* if mispredicted.
  On redirect, increment `epoch` and send new epoch with redirection.

- Wrong path: Discard any arriving instruction with the wrong epoch, and:
  - If instruction has `Rd`, "update Rd" to register file should just release reservation, not update Rd.
  - For memory STOREs (non-deferred), send "discard" message to store buffer.

- Committed instructions:
  - Memory STOREs (non-deferred): send "commit" message to store buffer.
  - Non-memory accesses (MMIO, deferred): perform the memory access, and handle ensuing traps, if any.

End