Learn RISC-V CPU Implementation and **BSV**

(**BSV**: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L18: RISC-V: Optimizing Drum and Fife

bluespec

Reminders

Please git clone: `https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design`
(git pull for latest version). Repsitory structure:

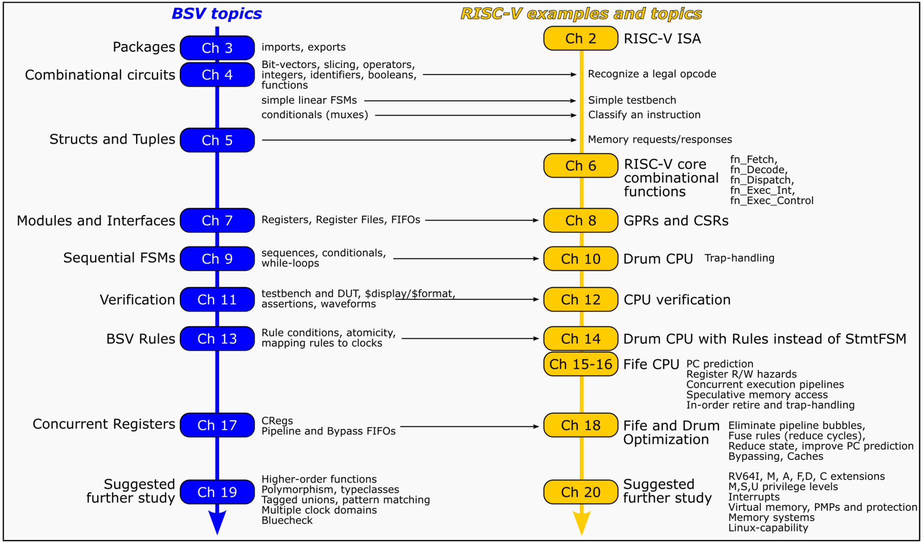
```
./Book_BLang_RISCV.pdf
  Slides/
      Slides_01_Intro.pdf
      Slides_02_ISA.pdf
      ...
  Exercises/
      Ex-03-A-Hello-World/
      Ex-03-B-Top-and-DUT/
      ...
  Code/
      src_Top/
      src_Drum/
      src_Fife/
      src_Common/
      ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- Slides and Exercise are numbered in sync with book Chapter numbers.
- For Exercises, please see Appendix E of the book. Some (not all) exercises have associated code in the `Exercises/` directory.

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



BSV topics

RISC-V examples and topics

Ch 2 RISC-V ISA

Packages Ch 3 imports, exports

Combinational circuits Ch 4 Bit-vectors, slicing, operators, integers, identifiers, booleans, functions → Recognize a legal opcode

simple linear FSMs → Simple testbench

conditionals (muxes) → Classify an instruction

Structs and Tuples Ch 5 → Memory requests/responses

Ch 6 RISC-V core combinational functions fn_Fetch, fn_Decode, fn_Dispatch, fn_Exec_Int, fn_Exec_Control

Modules and Interfaces Ch 7 Registers, Register Files, FIFOs → Ch 8 GPRs and CSRs

Sequential FSMs Ch 9 sequences, conditionals, while-loops → Ch 10 Drum CPU Trap-handling

Verification Ch 11 testbench and DUT, $display/$format, assertions, waveforms → Ch 12 CPU verification

BSV Rules Ch 13 Rule conditions, atomicity, mapping rules to clocks → Ch 14 Drum CPU with Rules instead of StmtFSM

Ch 15-16 Fife CPU PC prediction
Register R/W hazards
Concurrent execution pipelines
Speculative memory access
In-order retire and trap-handling

Concurrent Registers Ch 17 CRegs
Pipeline and Bypass FIFOs → Ch 18 Fife and Drum Optimization Eliminate pipeline bubbles,
Fuse rules (reduce cycles),
Reduce state, improve PC prediction
Bypassing, Caches

Suggested further study Ch 19 Higher-order functions
Polymorphism, typeclasses
Tagged unions, pattern matching
Multiple clock domains
Bluecheck

Ch 20 Suggested further study RV64I, M, A, F,D, C extensions
M,S,U privilege levels
Interrupts
Virtual memory, PMPs and protection
Memory systems
Linux-capability

Table of Contents

1. Reminders
2. General Comments
3. Optimizing Drum/Fife
4. Reducing misprediction penalty
5. Reducing register-hazard penalty
6. Final Comments

General Comments

What are we optimizing?

Three physical dimensions for optimization:

- Time (performance): how fast (wall-clock time) does the CPU execute an application?
- Space (area/resources): how much silicon area is taken up by the design (measured in gates, LUTs (lookup tables), BRAMs (block SRAMs), memories, DSPs, wiring congestion, *etc.*)?
- Energy: how much energy is consumed to execute an application?

These dimensions are not independent–an improvement in one may require a worsening in another (tradeoff).

Ultimately, a competitive product specification for a particular target market will define the acceptable boundaries for each dimension.

There is no easy way to estimate energy consumption other than actual measurement—run an application and measure the energe consumption. Optimizing energy consumption involves techniques such as lowering clock speeds and/or supply voltages dynamically during less critical periods.

Our focus here is on space/time tradeoffs.

CPU Performance

The time to execute an application is a product of multiple terms:

<exec time> = <total number of instrs> × 1/<clock-speed> × CPI

the units being:

seconds = instructions × seconds/cycle × cycles/instruction

"cycles/instruction" may vary depending on the type of instruction (*e.g.*, integer *vs.* floating point), and what precedes and follows an instruction (stalling due to register hazards, cache misses on fetch or DMem op, ...), so this should be considered to be an average.

"clock speed" may vary during application execution, in some CPU implementations; this should also be considered to be an average.

NOTE:

A design with a slower clock speed can have faster performance than a design with a higher clock speed if it can perform the computation with fewer instructions and/or with fewer cycles/instruction.

In the early 1990s Digital Equipment Corporation's first Alpha microprocessors had *much* higher clock speeds than the competition, but were often slower on benchmark applications because of the other factors.

To improve performance we can try to improve each of the product terms:

- Reduce number of instructions with different ISA (*e.g.*, RV32IM *vs.* RV32I; ARM *vs.* x86 *vs.* RISC-V)
- Reduce number of instructions with better coding) (*e.g.*, mergesort *vs.* bubblesort)
- Reduce number of instructions with better compiler
- Increase the clock speed
- Use fewer cycles to execute each instruction

Area *vs.* Clock-speed tradeoffs

Increasing clock speed:

- means each combinational path from one register's output to the next register's input can accommodate fewer gates, thereby doing less computation work.
- might require more stages in a pipeline, thereby increasing area due to inter-stage buffers and more inter-stage control circuitry (hand-shaking logic).

Sharing a hardware component (*e.g.*, sharing an adder for the ADD instruction, for Load/Store effective address calculation, for BRANCH/JUMP address calculation, *etc.*):

- Sharing a component can save area ... but ...
- requires a multiplexer on the input for the different input sources, and
- requires longer wiring to and from the component for its multiple uses, and
- requires control logic and possibly stalling logic to schedule the use of the component by its various users, ...
- all of which can result in higher area and longer delays (slower clock speed).

Transformations to “balance” a pipeline

The *critical* path is the longest-delay combinational path in the entire design. The clock speed has to be slow enough to accommodate this delay.

- Two adjacent pipeline stages whose combined delay is less than the critical path (in some other stage) can be *fused* into a single stage without changing the clock speed. This reduces the number of stages needed for any instruction.
- A stage with a long critical path can be *split* into two stages to permit a higher clock speed.

  After the split, the critical path may remain with this stage, or may now be due to some other stage.
- Combinational logic in a stage with a long critical path may be *moved* into the adjacent upstream or downstream stage. This will shorten the critical path of this stage while increasing the critical path of the adjacent stage.

  It may result in more or less inter-stage buffer space to hold the inter-stage value.

Such transformations may be iterated to *balance* the pipeline, *i.e.,* have roughly equal delays in each stage.

Pipeline traces for pipeline performance analysis

A *pipeline trace* is a trace output from the CPU that is much more detailed than an instruction trace (discussed in earlier chapters on functional verification).
On each clock, the CPU outputs the state of each pipeline stage:

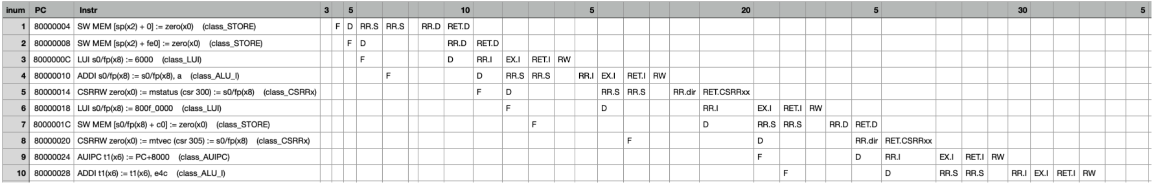
- Which instruction (if any) is currently in that stage, along with additional information carried along with the instruction.
- The stage state: stalled due to empty input; stalled due to full output; stalled due to scoreboard; ...

Pipeline traces can be analyzed:

- Manually: study the trace to detect issues
- Programmatically: an analysis program processes the trace looking for certain issues
- Visually (see next slide)

Visualization of pipeline traces

A Pipeline trace can be visualized as in the following example:



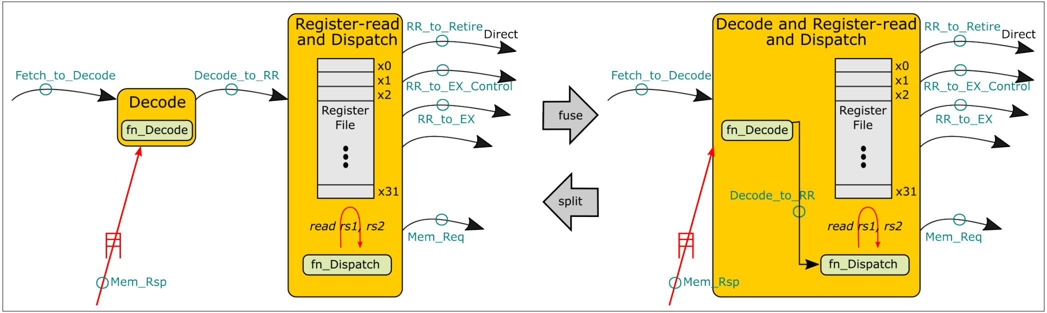
inum	PC	Instr	3		5					10					5					20					5					30					5
1	80000004	SW MEM [sp(x2) + 0] := zero(x0) (class_STORE)		F	D	RR.S	RR.S		RR.D	RET.D																									
2	80000008	SW MEM [sp(x2) + fe0] := zero(x0) (class_STORE)			F	D				RR.D	RET.D																								
3	8000000C	LUI s0/fp(x8) := 6000 (class_LUI)				F				D	RR.I	EX.I	RET.I	RW																					
4	80000010	ADDI s0/fp(x8) := s0/fp(x8), a (class_ALU_I)					F				D	RR.S	RR.S		RR.I	EX.I	RET.I	RW																	
5	80000014	CSRRW zero(x0) := mstatus (csr 300) := s0/fp(x8) (class_CSRRx)									F	D				RR.S	RR.S		RR.dir	RET.CSRRxx															
6	80000018	LUI s0/fp(x8) := 800f_0000 (class_LUI)										F				D				RR.I	EX.I	RET.I	RW												
7	8000001C	SW MEM [s0/fp(x8) + c0] := zero(x0) (class_STORE)											F							D	RR.S	RR.S		RR.D	RET.D										
8	80000020	CSRRW zero(x0) := mtvec (csr 305) := s0/fp(x8) (class_CSRRx)															F				D				RR.dir	RET.CSRRxx									
9	80000024	AUIPC t1(x6) := PC+8000 (class_AUIPC)																			F				D	RR.I	EX.I	RET.I	RW						
10	80000028	ADDI t1(x6) := t1(x6), e4c (class_ALU_I)																				F				D	RR.S	RR.S		RR.I	EX.I	RET.I	RW		

- The y-axis (going downwards is “inum”, the instruction number (serial number)
- The x-axis is time (each column is a clock tick).
- Each horizontal row represents one instruction traversing the pipeline, showing each stage (F=Fetch, D=Decode, EX.I=Execute Integer, RET.D=Retire DMem, RW=Register Write, ...).

Such a visualization dramatically highlights stalls (wasted cycles).
Once we see a stall, we can analyze that part of the pipeline trace in more detail to identify the reason for the stall and propose an improvement. The improvement may merely be improved compilation (*e.g.,* rearrange instructions to reduce the likelihood of scoreboard conflicts), or it could be an adjustment to the hardware design.

Optimizing Drum and Fife

Fusing/splitting stages: Decode and Register-Read-and-Dispatch



- The left-hand side depicts two stages. In Drum, a `Decode_to_RR` struct is communicated from function `fn_Decode` to `fn_Dispatch` *via* a register. In Fife, it is communicated *via* a FIFO.
- The right-hand size depicts a single *fused* stage. The value is communicated directly, by composing the functions directly (in hardware: just wires).

  The inputs and outputs of the fused stage are exactly the same the inputs and outputs of the original two stages.

Splitting a stage into two is the exact inverse of fusing. Please see book Chapter 18 for **BSV** code excerpts for fusing.

Fusing some Retire actions

In both Drum and Fife, exception handling is done in one place, by itself.

In Drum, it is the last, separate action in the FSM, `a_exception`, in module `mkCPU` in `src_Drum/CPU.bsv`.

In Fife, it is in rule `rl_exception` in module module `mkRetire` in `src_Fife/S5_Retire.bsv`.

In both cases, the exception is actually detected in earlier rules, which save values in three registers `rg_epc`, `rg_cause` and `rg_tval` which are then used by the exception handling action.

The exception-handling action can be fused directly into the earlier actions where the exception was detected, and the three registers can be eliminated.

This transformation may be less important because exceptions should be rare, and therefore saving these few ticks may not affect application performance very much.

Fife: saving FIFO resources

Section 16.3 with Figure 16.2 and Section 17.5.3 with Figure 17.7 describe how to connect Fife stages in a modular way, using a pair of FIFOs—a BypassFIFO in one stage (module) and a PipelineFIFO in the other, connected using `mkConnection` in the parent CPU module.

Logically this is a 1-element FIFO, but we use these two FIFOs to isolate the stages (separate rule scheduling, no combinational paths through the FIFO).

We could replace some FIFO pairs by a single FIFO, halving the register count for that connection. We lose isolation between stages and introduce combinational paths, but this may be harmless in selective cases.

Whether we use a PipelineFIFO or a BypassFIFO for this purpose depends on where it is used, because of the scheduling constraint. Usually we will need a PipelineFIFO in a forward path and a BypassFIFO in a reverse path because, for pipeline behavior we need to schedule a downstream rule before an upstream rule.

Fife: Reducing misprediction penalty

Fife: reducing misprediction penalty (1/3)

In Fife, the Fetch unit continually predicts the next PC so that it can continue fetching and feeding the pipeline. When it predicts incorrectly, this is detected in Retire, which redirects Fetch to the correct PC. In the meanwhile, Fetch has already fetched and fed a number of instructions into the pipeline, which must now all be discarded. The cycles spent on discarding these mispredicted instructions is called the *misprediction penalty*.

Optimization: Use CRegs in Fetch for PC and epoch

In Fetch, the PC and epoch are ordinary registers. The redirection action writes these registers, and the fetch action reads them, which can only happen on the next clock.

Optimization: By using CRegs for the PC and epoch, the redirection-writes and the fetch-reads can be done in the same clock, saving a tick.

Fife: reducing misprediction penalty (2/3)

Optimization: Eliminate FIFO on backward redirect path

The redirection message from Retire to Fetch passes through a FIFO.

Optimization: Eliminate this FIFO; let the Retire stage directly write into the PC and epoch registers in Fetch, saving a clock tick.

Caveat: less isolation between Retire and Fetch (longer combination paths, more rule-scheduling constraints).

Quicker reaction to redirection by Decode and Register-Read and Dispatch

A mispredicted instruction may stall on the scoreboard, spend many cycles in a long pipeline (memory, multiply/divide, floating-point), and reserve various resources (Rd register, slot in DMem store-buffer).

Optimization: When Retire redirects Fetch, also send the new epoch to Decode and Register-Read-and-Dispatch. They can immediately start discarding wrong-epoch instructions and/or convert them into no-ops, so they no longer reserve any resources or extra cycles.

Fife: reducing misprediction penalty (3/3)

Optimization: More accurate PC prediction

A PC-predictor is a function from PC to predicted-next-PC. Currently this function is very simple: just add 4 (predict PC+4).

This is not bad, but it mispredicts on BRANCH instructions where the branch is taken and on JAL and JALR instructions. It also fails on traps (jump to PC in CSR `mtvec`).

Improving the accuracy of PC-prediction is a vast topic: there are many, many papers on this topic, spanning many decades, in premier computer architecture conferences.

Optimization: "Branch Target Buffers" (BTBs) are associative tables, associating the PC of a BRANCH or jump instruction with the target PC it branched/jumped to. This information is updated with accurate information from redirection messages. The PC predictor consults this table and, if it finds a matching entry for the current PC, uses this value instead of PC+4 for the predicted PC.

Optimization: A "Return Address Stack" is a small "shadow stack" that mimics the saving and restoring of PCs on the real program stack on function calls and returns.

A JAL/JALR is likely a function call if it saves the current PC in the `ra` register. A JALR is likely a function return if it takes its target PC from the `ra` register. When these are recognized, they can push/pop PCs in the Return Address Stack, and these can be used for PC prediction.

Fife: Reducing register-hazard penalty

Fife: Reducing register-hazard penalty (1/3)

The *register-hazard penalty* is the number of cycles an instruction I2 may stall (idle wait) in the Register-Read stage because one or more of its input or output registers are "busy"—there is an earlier instruction I1 ahead of it in the pipeline that will be writing to one of those registers. This "stall" situation is detected and managed using the scoreboard.

Optimization: Saving a tick from register-update to dispatch

In stage `S3_RR_RW`

(A) Rule `rl_RR_Dispatch` reads and writes `rg_scoreboard` to manage register hazards, and reads one or two GPRs (in the `mkGPRs` module).

(B) Rule `rl_RW_from_Retire` reads and writes `rg_scoreboard` and writes zero or one GPRs (in the `mkGPRs` module).

Rule scheduling into clocks will therefore dictate that rule (A) cannot see the effects of rule (B) until at least one clock later.

Optimization: Use CRegs for both `rg_scoreboard` and for GPRs. This will allow rule (A) to see the results of rule (B) in the same clock, saving a tick.

Note: `mkGPRs` only needs a single `Bit#(XLEN)` CReg at the interface; the actual GPRs can remain an ordinary `RegFile`. See book Section 18.3.7.1 for how to do this.

Fife: Reducing register-hazard penalty (2/3)

Optimization: Eliminate FIFO on backward register-update path

The register-update message from Retire to Register-Read-and-Dispatch passes through a FIFO.

Optimization: Eliminate this FIFO; let the Retire stage directly write into the GPRs and scoreboard in Register-Read-and-Dispatch, saving a clock tick.

Caveat: less isolation between Retire and Register-Read-and-Dispatch (longer combination paths, more rule-scheduling constraints).

Fife: Reducing register-hazard penalty (3/3)

Optimization: Avoiding the stall on Rd

Currently, an instruction I2 stalls in Register-Read-and-Dispatch if it has the same Rd as some instruction I1 ahead of it in the pipeline, because that register will have been reserved in the scoreboard.

Optimization: Generalize the scoreboard from a single bit ("free" / "busy") to a small up-down counter (say 2 or 3 bits). In Register-Read-and-Dispatch, when dispatching an instruction,

- Stall it if the counter for its Rs1 or Rs2 is non-zero.
- If the counter for its Rd is not "saturated" (at its maximum value), increment the counter and allow the instruction to proceed (do not stall).

In Register-Read-and-Dispatch, when we receive a register update message from Retire, decrement the counter for Rd.

Use a CReg for each register's scoreboard up-down counter, as described in Chapter 17, to allow the register-update and dispatch in the same cycle.

Final comments on Optimization

This chapter has limited itself to optimizations that do not fundamentally change the microarchitecture of Fife and Drum, and has avoided discussing more radical changes (to superscalar, out-of-order, *etc.*). The latter are briefly discussed in Chapter 20. As such, the optimizations discussed here are relatively local and non-disruptive.

The major ideas are:

- Fusion and splitting of stages and actions
- Using leaner FIFOs
- Improving accuracy of PC-prediction, and reacting more quickly to misprediction
- Sharing circuits (less area, but more MUXes and control logic) and un-sharing
- Using CRegs to save clock ticks
- Eliminating the stall on Rd

End