# Learning RISC-V Implementation and Hardware Design with the High-Level Hardware Design Language BSV

Rishiyur S. Nikhil
Bluespec, Inc.




**DRAFT:** February 29, 2024

# Contents


**1 Introduction** **1-1**

1.0.1 Drum and Fife, the two RISC-V implementations designed in this book . . . 1-3

1.0.2 Drum and Fife source codes . . . 1-4

1.0.3 Additional Resources . . . 1-4

**2 Overview of the RISC-V ISA** **2-1**

2.1 Introduction . . . 2-1

2.2 What is an ISA? . . . 2-1

2.3 Why choose RISC-V? . . . 2-3

2.4 Overview of the RISC-V ISA . . . 2-4

2.5 Instruction encodings . . . 2-6

2.6 Unprivileged ISA RV32I . . . 2-7

2.6.1 "Upper Immediate" instructions LUI and AUIPC . . . 2-7

2.6.2 Conditional BRANCH instructions . . . 2-9

2.6.3 LOAD and STORE memory-access instructions . . . 2-10

2.6.4 Register-Register Arithmetic and Logic instructions . . . 2-10

2.6.5 Register-Immediate Arithmetic and Logic instructions . . . 2-11

2.6.6 Unconditional Jump instructions . . . 2-11

2.6.7 FENCE . . . 2-12

2.7 Traps due to illegal instructions and other exceptions, and CSRs . . . 2-12

2.7.1 ECALL and EBREAK instructions, and Interrupts . . . 2-14

2.7.2 CSRRxx instructions . . . 2-14

2.8 RV64I differences from RV32I . . . 2-15

2.9 Continued Evolution of the RISC-V ISA (with your contribution?) . . . 2-16

**3 RISC-V interpreters: the Design Space from Software Functional Simulators to High-Performance Hardware** **3-1**

3.1 The RISC-V designs in this book . . . 3-2

3.2 Abstract algorithm for interpreting an ISA . . . 3-3

3.2.1 Memory latency and split-phase memory transactions . . . 3-4

3.3 Plan for the order in which we tackle topics . . . 3-5

**4 BSV: Combinational circuits for the RISC-V step functions** **4-1**

4.1 Introduction . . . 4-1

4.2 Bit Vectors . . . 4-2

4.2.1 Built-in Operators on Bit Vectors . . . 4-2

4.3 Integer types . . . 4-3

4.4 Hexadecimal and Binary Notation for literal integers . . . 4-4

4.5 Boolean values . . . 4-4

4.5.1 Caution: `Bool` and `Bit#(1)` are different types . . . 4-5

4.5.2 Example: recognizing legal RISC-V BRANCH instructions . . . 4-5

4.5.3 Combinational circuits and primitives . . . 4-6

4.6 Functions . . . 4-7

4.6.1 Pure functions vs. functions with side-effects (`Action`, `ActionValue`) . . . 4-7

4.6.2 Combinational circuits = "doesn't have `Action` or `ActionValue` type" . . . 4-8

4.6.3 Using `ActionValue` on pure functions for `$display` debugging . . . 4-9

4.7 A small testbench to test our code . . . 4-9

4.8 `enum` types . . . 4-11

4.8.1 `deriving (Bits)` . . . 4-12

4.8.2 `deriving (Eq)` . . . 4-12

4.8.3 `deriving (FShow)` . . . 4-12

4.9 Syntax of Identifiers . . . 4-12

4.10 Syntax of comments . . . 4-13

4.11 if-then-else statements and hardware multiplexers . . . 4-13

4.11.1 Parallel multiplexers and MUX synthesis . . . 4-15

4.12 Sharing code for RV32 and RV64 *via* parameterization . . . 4-16

4.12.1 Numeric types . . . 4-17

4.12.2 Type synonyms . . . 4-17

4.12.3 The numeric value corresponding to a numeric type . . . 4-18

4.12.4 Conditional compilation . . . 4-18

**5 BSV: Struct types, and RISC-V: Memory requests and responses** **5-1**

5.1 RISC-V: structs communicated between steps . . . 5-1

5.2 BSV: `struct` types . . . 5-2

5.2.1 Creating struct values . . . 5-4

5.2.2 BSV: Don't-care values . . . 5-4

5.2.3 Selecting struct fields . . . 5-5

5.2.4 Updating struct fields using assignment . . . 5-5

5.3 RISC-V: Memory Requests and Responses; IMem and DMem . . . . . 5-5
5.3.1 Separation of IMem and DMem (Harvard Architecture) . . . . . 5-6
5.3.2 RISC-V: Memory Requests . . . . . 5-6
5.3.3 RISC-V: Address Alignment . . . . . 5-7
5.3.4 RISC-V: Memory Responses . . . . . 5-8

**6 RISC-V: Core functions for RISC-V ISA execution (used in Drum and Fife) 6-1**
6.1 Introduction . . . . . 6-1
6.2 The function `fn_Fetch` . . . . . 6-1
6.3 The `fn_Decode` function . . . . . 6-3
6.4 The `fn_Dispatch` function after reading input registers . . . . . 6-7
6.5 The `fn_EX_Control` function . . . . . 6-11
6.6 The `fn_EX_Int` function . . . . . 6-13
6.7 No separate functions for Execute DMem and Retire . . . . . 6-17

**7 BSV: Modules and Interfaces: Registers, Register Files and FIFOs 7-1**
7.1 Introduction . . . . . 7-1
7.2 Modules: state, interfaces and behavior . . . . . 7-1
7.2.1 Internal behavior (*rules*) . . . . . 7-2
7.2.2 Interface declarations . . . . . 7-2
7.2.3 Module declarations . . . . . 7-3
7.2.4 Module instantiation and method invocation . . . . . 7-3
7.3 BSV Library Modules: Registers . . . . . 7-4
7.3.1 `Reg#(t)`, the register interface from the BSV library . . . . . 7-4
7.3.2 Registers are strongly typed . . . . . 7-4
7.3.3 `mkReg(`$v$`)`, a register module (consructor) from the BSV library . . . . . 7-5
7.3.4 Syntactic shorthands for register access . . . . . 7-5
7.4 BSV Library Modules: Register files . . . . . 7-6
7.4.1 The register file interface `RegFile#(index_t,data_t)` from the BSV library 7-6
7.4.2 `mkRegFileFull`, a register file module (constructor) from the BSV library . . 7-7
7.5 BSV Library Modules: FIFOs . . . . . 7-7
7.5.1 `FIFOF#(`$t$`)`, the FIFO interface from the BSV library . . . . . 7-7
7.5.2 `mkFIFOF`, a FIFO module (constructor) from the BSV library . . . . . 7-8
7.5.3 FIFOs are strongly typed . . . . . 7-8
7.5.4 Semi-FIFO interfaces for each end of a FIFO . . . . . 7-9
7.5.5 Interface-transformer functions . . . . . 7-9
7.5.6 Connecting FIFOs . . . . . 7-10
7.5.7 `mkPipelineFIFOF` and `mkBypassFIFOF`: constructors from the BSV library . . 7-11
7.6 Polymorphic and Monomorphic Types . . . . . 7-11
7.6.1 Polymorphic Modules and Synthesizability into Verilog . . . . . 7-12

**8 RISC-V: Modules for GPRs and CSRs** **8-1**

8.1 Introduction . . . 8-1

8.2 A register file for RISC-V GPRs, with special treatment of `x0` . . . 8-1

8.2.1 Inlined *vs.* separate module `mkRISCV_GPRs` . . . 8-2

8.3 A register file for RISC-V CSRs . . . 8-3

**9 BSV: FSMs** **9-1**

9.1 Introduction . . . 9-1

9.1.1 Sequential FSMs, Concurrent FSMs, and Digital Hardware . . . 9-2

9.2 Rules and StmtFSM in BSV . . . 9-3

9.3 Actions and the `Action` type . . . 9-3

9.3.1 `Action` blocks: composing actions into larger actions . . . 9-4

9.3.2 Binding names in `Action` blocks . . . 9-4

9.4 `StmtFSM`: sequences of actions . . . 9-5

9.5 `StmtFSM`: conditionals (if-then-else) . . . 9-5

9.6 `StmtFSM`: while-loops . . . 9-6

9.7 `StmtFSM`: pausing until some condition holds . . . 9-6

9.8 `StmtFSM`: `mkAutoFSM`: a simple FSM module constructor . . . 9-6

9.9 `StmtFSM`: many more features . . . 9-7

**10 RISC-V: the Drum unpipelined CPU (an FSM)** **10-1**

10.1 Introduction . . . 10-1

10.2 The Drum CPU module interface . . . 10-1

10.3 The Drum CPU module . . . 10-2

10.4 Help-functions for the Drum CPU module behavior . . . 10-4

10.5 The Drum CPU module behavior . . . 10-5

10.5.1 FSM action for Fetch . . . 10-6

10.5.2 FSM action for Decode . . . 10-6

10.5.3 FSM action for Dispatch . . . 10-7

10.5.4 FSM actions for Execute and Retire . . . 10-8

10.5.5 FSM actions for exceptions . . . 10-12

10.6 Conclusion . . . 10-12

10.6.1 But Drum code looks just like C!? Why not code it in C? . . . 10-13

**11 BSV: Packages, Files and Imports; System Top-Level** **11-1**

11.1 Packages and Files . . . 11-1

11.2 Importing C; Fife/Drum memory system . . . 11-1

11.3 `Top.bsv` instantiating `CPU.bsv` and `Mems_Devices.bsv` . . . 11-1

**12 RISC-V: the Fife pipelined CPU: Principles** **12-1**
12.1 Introduction . . . 12-1
12.2 Keeping the Fetch Stage Working with PC Prediction and Epochs . . . 12-2
12.2.1 PC Prediction in the Fetch Stage . . . 12-2
12.2.2 Identifying and Flushing Wrong-path Instructions . . . 12-3
12.2.3 Terminology: Speculative Instructions and Commits . . . 12-4
12.2.4 Speculative instructions should not have any side-effects . . . 12-4
12.3 Managing Register Read/Write Hazards with a Scoreboard . . . 12-4
12.3.1 Releasing Scoreboard Reservations for Uncommitted Instructions . . . 12-6
12.3.2 Bypassing . . . 12-6
12.4 Retiring outputs of the Execute Stages in Order with Tags . . . 12-7
12.5 Allowing Memory Ops to be Pipelined, with a Store Buffer . . . 12-7
12.5.1 What about LOADs and STOREs to non-memory-like devices (MMIO)? . . . 12-8
12.6 The Retire Stage . . . 12-9

**13 RISC-V: the Fife pipelined CPU: BSV code** **13-1**
13.1 Introduction . . . 13-1
13.2 The Fife top-level CPU module . . . 13-1
13.3 Fife: the Fetch stage . . . 13-3
13.4 Fife: the Decode stage . . . 13-5
13.5 Fife: RR-RW module (Register-Read, Dispatch, Register-Write) . . . 13-6
13.5.1 BSV: Vectors for the Scoreboard . . . 13-7
13.5.2 The RR-RW module . . . 13-8
13.6 Fife: the Execute Control stage . . . 13-11
13.7 Fife: the Execute Integer Ops stage . . . 13-12
13.8 Fife: the Execute Memory Ops stage (speculative DMem) . . . 13-13
13.9 Fife: the Retire stage . . . 13-13
13.9.1 Common facilities used by many rules . . . 13-16
13.9.2 Rule to discard wrong-path instructions . . . 13-17
13.9.3 Rules to retire instructions direct from RR-RW . . . 13-17
13.9.4 Rules to retire instructions from the Execute Control path . . . 13-18
13.9.5 Rules to retire instructions from the Execute Int path . . . 13-19
13.9.6 Rules to retire instructions from the Execute DMem path . . . 13-20

**14 BSV: Rules, Clocks and Rule Scheduling** **14-1**
14.1 Rules, Actions and clocks . . . 14-1
14.2 Rule semantics . . . 14-1
14.3 Rules scheduling. Example: Counter with `incr` and `decr` methods . . . 14-1
14.4 CRegs . . . 14-1
14.5 Implementing mkPipelineFIFOF and mkBypassFIFOF with CRegs . . . 14-1
14.6 Saving a cycle for Redirect and RW/Scoreboard-write . . . 14-1

**15 BSV: Suggested further study** **15-1**

15.1 Alternative simulators: Bluesim, other Verilog sims . . . 15-1
15.2 Static elaboration vs. execution . . . 15-1
15.3 First-class modules . . . 15-1
15.4 Polymorphism . . . 15-1
15.5 Typeclasses. Conversion of Integer literals. Bits/pack/unpack . . . 15-1
15.6 Bluecheck . . . 15-1
15.7 Tagged unians, pattern-matching . . . 15-1
15.8 BH alternative syntax . . . 15-1

**16 RISC-V: Suggested further study** **16-1**

16.1 Advanced branch prediction . . . 16-1
16.2 Memory systems: TCMs, Caches, PMPs . . . 16-1
16.3 M Extension . . . 16-1
16.4 A extention . . . 16-2
16.5 F and D Extensions . . . 16-2
16.6 C Extension . . . 16-2
16.7 Virtual Memory . . . 16-2
16.8 Performance measurement . . . 16-2
16.9 Testing . . . 16-2
16.10 Interrupts . . . 16-2
16.11 Linux and server-class capability . . . 16-3
16.11.1 Hypervisor support . . . 16-3
16.11.2 RISC-V ISA Formal Specification . . . 16-3

**A Resources: Documents and Tools** **A-1**

A.1 GitHub . . . A-1
A.2 RISC-V ISA (Instruction Set Architecture) Specifications . . . A-1
A.3 RISC-V Assembly Language Manuals . . . A-2
A.4 RISC-V GNU tools, including `riscv-gcc` compiler . . . A-2
A.5 BSV . . . A-2
A.5.1 "BSV By Example" book (free downloadable PDF) . . . A-3
A.5.2 **BSV** Tutorial . . . A-3
A.5.3 MIT Course Material . . . A-4
A.5.4 University of Cambridge Examples . . . A-4
A.5.5 *bsc* download and installation; *bsc* and **BSV** manuals . . . A-4
A.6 Verilator (or other Verilog simulator) . . . A-4
A.7 Amazon AWS . . . A-5
A.8 Xilinx Vivado . . . A-5
A.9 RISC-V textbooks . . . A-6

**B Why BSV?** **B-1**

B.1 Why BSV instead of some other Hardware Design Language? . . . . . B-2

B.1.1 A better computational model . . . . . B-2

B.1.2 Modern language features . . . . . B-3

B.1.3 Comparison with C++-based High Level Synthesis . . . . . B-4

**C Glossary** **C-1**

**Index of BSV topics** **INDEX-BSV-1**

**Bibliography** **BIB-1**

# Chapter 1

# Introduction

"Digital Design" and "CPU Design" (or "Computer Architecture") are traditionally taught separately, usually in that order, with separate textbooks. Digital Design is usually taught using one of the traditional hardware design languages Verilog, SystemVerilog or VHDL, and often makes use of small, often artificial examples. CPU Design is often taught without actually designing hardware, relying instead on textbooks, abstract schematics, and simulators implemented in software.

This book takes a different approach: we learn about simple CPU architectures by designing them with a modern Hardware Design Language (HDL) called BSV, learning Digital Design as an ongoing, intertwined accompanying topic. Each Digital Design example will be taken directly from the CPU Design, so that the example's use-case (context) is always perfectly clear, and the reader always has a clear sense of the purpose of the example.

The CPUs we design here will execute instructions from the RISC-V Instruction Set Architecture (ISA), which is an industrial-strength ISA (with many commerical implementations). Our designs will be simple (typical of small, embedded systems and micro-controllers, not laptops/workstations or servers).

Figure 1.1 shows the broad range of topics in which a CPU designer needs to engage. Some of the early topics are not RISC-V implementation-specific:

- The first step is to understand the RISC-V ISA itself. What are RISC-V instructions, how are they coded in bits, and what do they mean? This topic is not a focus of this book (for which there are plenty of textbooks and other educational materials available), but understanding the ISA is of course a prerequisite to informing our design, so we provide a brief overview in Chapter 2.

  The RISC-V ISA has many options; our focus in this book will be on a small "standard" subset that is adequate for small embedded systems:

  - The so-called "RV32I" subset from the RISC-V Unprivileged ISA spec: basic integer arithmetic and logic operations; branch and jump; load and store.
  - A few elements from the RISC-V Privileged ISA spec for handling exceptions: Control and Status Registers (CSRs), traps and trap-handling.

- In order to run actual RISC-V programs on our implementations, we need to understand how to use the *riscv-gcc* compiler to compile C and RISC-V Assembly Language

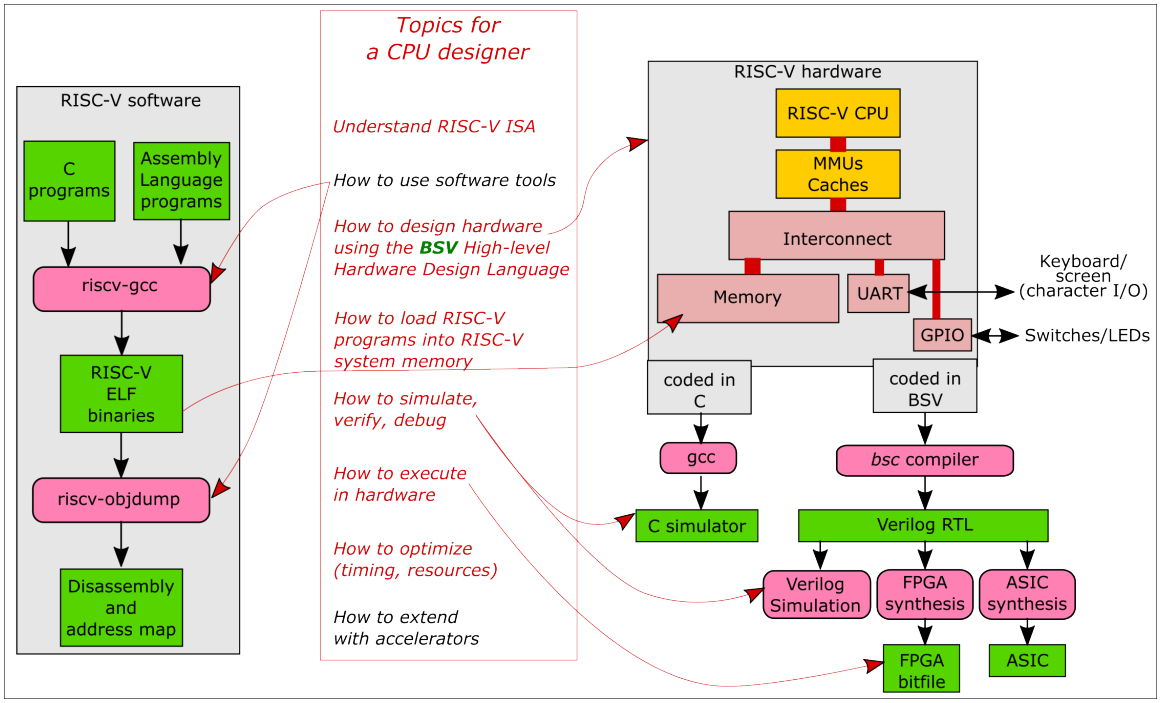



Figure 1.1: Topics covered in this book (in red text in red box)

programs into RISC-V binaries (so-called "ELF" files). Another useful tool is *riscv-objdump*, which can disassemble the binary back into assembly-language text. This is useful for debugging our implementation, so that we can understand execution instruction-by-instruction, and diagnose anything that goes wrong.

How to install and use these tools is not a focus of this book but of course we need to use these tools to produce programs to run on our implementations.

So, far, all this is not implementation-specific, *i.e.,* it is generic information about RISC-V. The following topics dive into implementations.

- A RISC-V CPU and system can be *modeled* in a simulator coded in C (say). Such a C-based simulator is compiled (with *gcc*, say) and run like any other C program. We will not be discussing this much in this book.
- We will code our hardware design in the BSV HL-HDL. We will use BSV not just for the CPU itself, but also for the "system" components around it: an interconnect, Memory, UART and GPIO.
- We will use the *bsc* compiler to translate our BSV code into Verilog RTL.
- We will learn how our Verilog RTL can directly be simulated in a Verilog simulator. We will use the free, open-source "Verilator" simulator, but you can also run it on any other Verilog simulator, available from a number of providers.

  This will provide an exact, cycle-by-cycle accurate simulation of the very same design that we'll run later on an FPGA. This is invaluable for debugging the hardware design,

because the turnaround time to fix a problem and run a new simulation is very short (minutes) compared to creating a new version for an FPGA (several hours).

Of course, Verilog simulation will run much more slowly (10,000x or more slower) compared to an FPGA, and so is useful primarily for early debugging and analysis of the design, running on small RISC-V programs.

- When we execute our Verilog RTL hardware design in Verilog simulation (where the hardware design itself is executing a RISC-V binary program), it will produce a trace file describing events during the simulation. We will learn how to analyze these traces to identify bugs and bottlenecks in our design, from which we can correct design errors and possibly improve performance.
- We will discuss how to process our Verilog RTL through an FPGA synthesis tool to create an FPGA bitfile which can then be loaded into an FPGA and executed.

  Although it can be synthesized and run on a number of FPGAs from different vendors, in this book we'll discuss how to build and run it for an FPGA on the Amazon AWS cloud.
- Our Verilog RTL can also be processed through ASIC synthesis tools targeting ASIC fabrication. We will not be dicsussing this much in this book.

### 1.0.1 Drum and Fife, the two RISC-V implementations designed in this book

We will create two hardware RISC-V designs, called Drum and Fife:

- Drum is a *non-pipelined* implementation which will familiarize us with all the basic concepts and flows (the RISC-V ISA, preparing and running a RISC-V binary to run on the design, analysing traces), without being distracted by the complexities of pipelineing for high performance.
- Fife is a 5-6 stage pipelined design, which is a microarchitectural change focused on higher performance (speed) than Drum. The design includes simple branch-prediction, register read/write hazard management, in-order retirement of instructions, and speculative stores to memory using a store-buffer.

The two designs will share a large part of the BSV code that implements the essential semantic functionality of the RISC-V ISA. By discussing this shared code in the simpler Drum, the Fife chapters can focus purely on the new issues raised by pipelining (and which are, in fact, not RISC-V specific, but common to all pipelined CPU designs).

Drum and Fife execute exactly the same binaries; the only difference will be in Fife's superior performance (speed).

As we work through the two designs, we will concurrently learn how to code in BSV, the HL-HDL for our designs. BSV is a modern, high-level HDL taking inspiration from modern software programming languages, in particular the Haskell functional programming language and a class of formal specification languages for concurrent programming (including Term Rewriting Systems, Unity, TLA+, and Event-B). BSV is not just for CPU design; like Verilog and SystemVerilog, it is a "universal" language for any digital design, whether related to CPUs or not. Appendix B has a more detailed justification of our choice of BSV over other hardware design languages like Verilog, SystemVerilog, VHDL or Chisel.

### 1.0.2 Drum and Fife source codes

The full source codes (BSV) for Drum and Fife are included with this book. Excerpts of that code are taken, as-is, for inclusion in this book. Excerpts look like this:

src_Common/CPU_IFC.bsv: line 27 ...

```
1 interface CPU_IFC;
2    method Action init (Initial_Params initial_params);
3
4    interface FIFOF_O #(Mem_Req) fo_IMem_req;
5    interface FIFOF_I #(Mem_Rsp) fi_IMem_rsp;
6    ...
7    interface FIFOF_O #(Mem_Req) fo_DMem_req;
8    interface FIFOF_I #(Mem_Rsp) fi_DMem_rsp;
9 endinterface
```

The label on the top border of the box indicates file (`src_Common/CPU_IFC.bsv`) from which this has been extracted, and the starting line number (27) in the file. The "`...`" indicates we have *elided* some lines from the source file which are not directly relevant to the current discussion.

We recommend, as you read the book, that you also keep open a text-editor, in which you can simultaneously view the actual sources.

### 1.0.3 Additional Resources

Chapter 15 has suggestions for further study of BSV.

Chapter 16 has suggestions for further study of RISC-V and CPU design.

Appendix A has a detailed listing of resources (documents and software tools) needed for this book, and for further reading.

This book also contains a Glossary of terms and abbreviations, and a detailed index.

# Chapter 2

# Overview of the RISC-V ISA

## 2.1 Introduction

This entire chapter can safely be skipped by those already familiar with RISC-V concepts, or who learn it elsewhere (there are many alternate resources on the web). This chapter contains only generic information about ISAs and the RISC-V ISA in particular. It can be read as a general introduction to the RISC-V universe; none of this is specific to this book.

## 2.2 What is an ISA?

The acronym "ISA" stands for "Instruction Set Architecture". Figure 2.1 illustrates the role of an ISA as an intermediary between hardware (CPU implementations) and software (which runs on the CPU hardware).

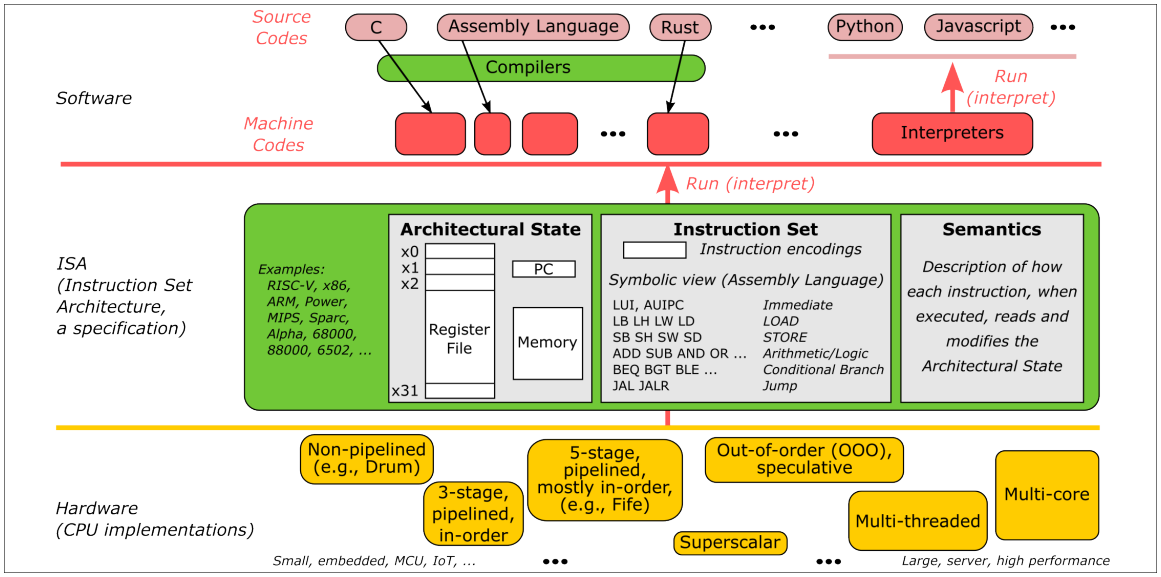



Figure 2.1: What is an ISA?

An ISA, *per se*, is neither software nor hardware. It is merely a *specification*, a document defining three things:

- *Architectural State*: the registers visible to the instructions, such as the program counter (PC) and a register file containing, say, 32 registers named x0 through x31.

  Typically (nowadays), the architectural state includes a *byte-addressed* memory (a memory where an *address* identifies a single byte, and where individual bytes may be read and written.

- An *Instruction Set*: a collection of instructions. For each instruction, the ISA specifies how it is encoded in bits. The ISA may also specify a way of writing an instruction in symbolic text, which we call Assembly Language. Instructions are typically grouped in classes, such as Immediate, LOAD/STORE, Arithmetic and Logic, Conditional Branches, Jumps, System Instructions, and so on.

- *Semantics*: a description, for each instruction about *how* it executes—what architectural state it observes, and what architectural state it updates, and how. For example, a Conditional Branch instruction typically observes two registers in the register file, compares them, and updates the PC depending on the comparison result.

  The overal semantics of a program being executed is just a sequential composition of individual instruction semantics.

The full ISA may be described in ordinary text prose and diagrams, or in a semi-formal or formal language. For example, the RISC-V ISA is described both in text prose and diagrams, and in the formal language "Sail".

Beneath the ISA level are actual *implementations* of the ISA. These range from software simulators of the ISA to silicon implementations. Silicon implementations typically vary widely in their *micro-architecture* features, such as pipelining, in-order or out-of-order, speculation, superscalarity, multi-threading, multi-core. The choice of micro-architecture is typically based on a particular target market, trading-off cost, performance (speed), energy consumption, capabilities (embedded software to full server OS with network and storage stacks), silicon technology (FPGA *vs.* ASIC, ASICs with various silicon feature sizes and techniques), and so on. Variations in micro-architectures provide product differentiation.

Each implementation runs (interprets) machine code, *i.e.,* an encoding of instructions in memory. The machine codes are typically produced by compilers, which are themselves machine code programs that transform source codes into machine codes. Some machine codes are themselves software interpreters for so-called interpreted languages such as Python, Javascript, Java, Scheme, Lisp, and so on.

A crucially important point is that *all the implementations of an ISA should respect the semantics of the ISA as specified in the ISA definition.* They can (and do) play all kinds of micro-architectural tricks under the covers, but the result of executing any program should be explainable purely based on the ISA specification.

Thus, and ISA is a *contract* or *API* between software and hardware implementations. People writing software, people writing compilers for software, people writing interpreters for interpreted languages, *etc.* need only to understand and refer to the ISA specification to do their job. They do not need to know about the specific hardware implementation

on which it will eventually run. This enables *software portability*, *i.e.,* a machine code program for an ISA should be able to run, without change, on *any* implementation of the ISA, including future next-generation laptops/servers for the ISA.

NOTE: In the rarer cases where the "bleeding edge" of software performance really matters, human coders and compilers may produce different machine codes for specific implementations of the same ISA, in order to exploit particular quirks of each implementation's micro-architecture such as branch-prediction or register-hazard penalties.

Examples of ISAs include:

- *RISC-V*: Open ISA (not proprietary). Silicon implementations from various vendors, worldwide.
- *x86*: Proprietary. Silicon Implementations from Intel and AMD, primarily, with names like Xeon, Core i9, 13th Generation Core, Alder Lake, Raptor Lake, and so on.
- *ARMv8*, *ARMv9*: Proprietary. Implementations from ARM licensees like Apple, Samsung, Broadcom and others.
- *Power*: Proprietary. Implementations from IBM and other licensees.
- *Sparc*: Proprietary. Implementations originally from Sun Microsystems, nowadays from Oracle, and also from licensees such as Fujitsu.
- *MIPS*: Proprietary. Implementations from MIPS and other licensees.
- Other famous, but now defunct ISAs: *Alpha* (from Digital Equipment Corp./Compaq/HP), *Itanium* (from Intel, HP), *68000* and *88000* (from Motorola), ...

## 2.3 Why choose RISC-V?

In the list of ISAs above, except for RISC-V, all the other ISAs are *proprietary*, *i.e.,*, in order to produce and sell a silicon implementation it is necessary to obtain a license (permission) from the company that "owns" the ISA. These licenses can be very expensive, in the thousands of dollars or more.

The RISC-V ISA is owned by RISC-V International ("RVI"), a non-profit corporation based in Switzerland (https://riscv.org). As of Fall 2023, RVI claimed a growing membership, including over three thousand corporate, government, university, and individual members from over 70 countries.

The ISA is "open" in that no prior permission or license from RVI is needed in order to produce and sell implementations of the ISA. If a *commercial* product is claimed publicly to implement the RISC-V ISA, the vendor needs to get official certification from RVI that it indeed does so (it must pass a battery of certification tests), but this is a one-time certification, quite different from a production license. Non-commercial products (from universities, research labs, hobbyists, *etc.*) do not need any such certification.

Separate from these commercial concerns, there are also concerns about quality of an ISA and the richness of the ecosystem supporting an ISA. The RISC-V ISA is attractive on these dimensions as well.

The RISC-V ISA was originally designed at University of California, Berkeley, by a team of researchers who have half a century of deep knowledge and experience on ISAs, computer

architectures, computer systems, and ecosystem software (the Sparc ISA also came out of Berkeley in the 1980s). As such, the RISC-V ISA can be seen as a new, clean-slate design that incorporates all the lessons learned from all previous ISAs dating all the way back to the 1950s.

An ISA is useless without a strong *ecosystem* supporting the ISA. This includes artefacts such as compilers, programming language implementqtions, debuggers, test and verification infrastructure, embedded operating systems, real-time operating systems, workstation and server-class operating systems, boot loaders, device drivers, and so on. It also includes services, such as tutorials, books, courses, and training materials (this book can be seen as a contribution). The RISC-V ecosystem is already quite rich, and it grows daily precisely because of the open nature of the ISA, enabling thousands of contributors worldwide to participate in the effort.

The openness of the RISC-V ISA also enables entrepreneurs and researchers to attempt *innovations* in micro-architecture and design and production techniques, something which is not practically feasible with a proprietary ISA.

The RISC-V ISA is unusually *modular* (more details in Section 2.4). It consists of a *very* small "base" Integer ISA and a number of optional standard extensions (such as Integer Multiply/Divide, floating point, Atomics, and Compressed instructions), and a systematic way of substituting or adding new, non-standard, proprietary extensions. This makes it easier for an entrepreneur to "tune" the ISA for their proprietary impliementation aimed at a target market with specific requirements..

There is also a security dimension to RISC-V's attractiveness. When a customer buys a silicon implementation of an ISA, they have to trust that neither the vendor nor the supply chain inserted any secret "back-doors" by which others can monitor what the processor is doing or, worse, disable it or program it remotely. The RISC-V ISA being open, it is more possible for a customer to develop their own trusted supply chains for silicon implementations.

Silicon implementations of RISC-V are already availble from dozens of vendors located in several countries worldwide. These supply chains are only likely to grow more diverse. Many production-ready, competitive RISC-V designs are available in free and open-source form.

## 2.4 Overview of the RISC-V ISA

NOTE:

More detailed information on the topics of this section can be found in:

- RISC-V ISA specification documents. Please see Appendix A.2 for links.
- Formal specification of the RISC-V ISA, written in the Sail language. Please see Appendix A.2 for links.
- RISC-V Assembly Language manuals Please see Appendix A.3 for links.

The RISC-V ISA is designed to be highly modular. Figure 2.2 shows the major components. The foundations (*base* integer ISA) are RV32I and RV64I, for implementations with 32-bit and 64-bit register widths, respectively. Technically these are two separate ISAs, but all but two of the RV32I instructions have identical counterparts in RV64I, so one can think of RV64I as a superset of RV32I, as suggested in the diagram. RV32I has just 40 instructions. RV64I slightly modifies two of them and adds a few more instructions.

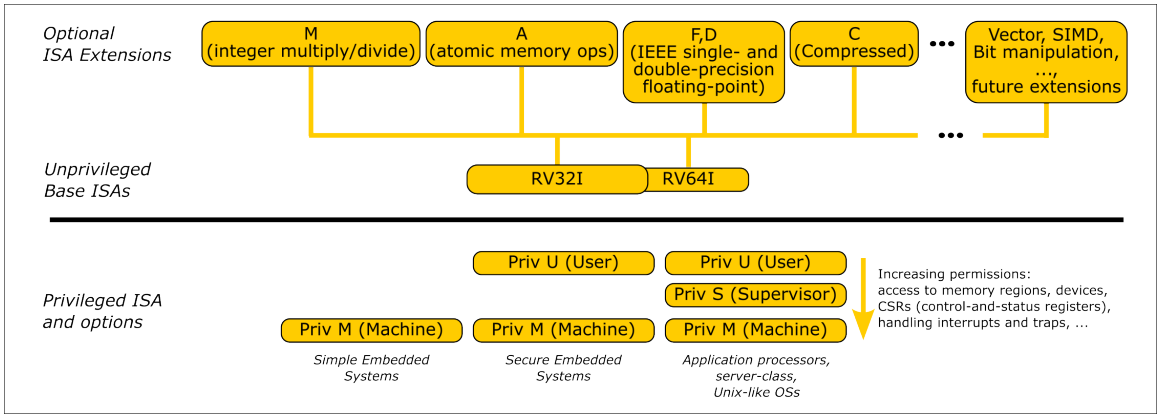



Figure 2.2: Modularity of the RISC-V ISA

To the base ISA one can add several standard optional ISA extensions:

- M: a few instructions for integer multiplication and division.
- A: a few instructions for atomic memory operations (atomic read-modify-write of locations in memory).
- F,D: several instructions for IEEE single- and double-precision floating point operations.
- C: several so-called "compressed" instructions, which are only 16-bits wide, for applications where it is important for the code size to be small.
- Vector extension for vector arithmetic for scientific, AI and high-performance computing.
- Other extensions like SIMD (Single Instruction, Multiple Data), Bit Manipulation, useful in image processing, cryptography, *etc.*

The standard Privileged ISA is shown in the bottom half of the diagram. This consists of three "privilege" levels U (User, low), S (Supervisor) and M (Machine, high), with increasing capabilities with respect what aspects of the architectural state are visible and updatable.

The transition into the Privileged ISA only happens through a few carefully managed gateways. Thus, the whole standard Privileged ISA can easily be substituted with some other, non-standard, Privileged ISA, should the implementor find that beneficial.

Simple RISC-V implementations for very small embedded systems may implement only one privilege level (M). Since there is only one privilege level, there are no relative protections—all code can access all architectural state.

Slightly more secure RISC-V implementations for embedded systems may implement two privilege levels (M and U). Now, code running at U privilege can be prevented from accessing (and damaging) certain parts of architectural state such as devices, regions of memory, *etc.*

Most medium- to large-size systems implement all three privilege levels, M, S and U. Typically, user code runs at U privilege; operating systems such as Linux run at S privilege, and low-level device-access codes run at the highest (M) privilege. When S is implemented, code running at both U and S privilege levels can also use *virtual memory*.

## 2.5 Instruction encodings

All RISC-V instructions are encoded in 32 bits (except for the C extension, where they are encoded in 16 bits). Although there are hundreds of instructions if we count RV32I, RV64I, M, A, F, D, C, and Privileged ISA, they are all encoded in just a few 32-bit formats, shown in the Unprivileged spec, page 130, reproduced here in Figure 2.3. The labels on the



| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |
| imm[12\|10:5] | | rs2 | rs1 | funct3 | imm[4:1\|11] | opcode | B-type |
| imm[31:12] | | | | | rd | opcode | U-type |
| imm[20\|10:1\|11\|19:12] | | | | | rd | opcode | J-type |

Figure 2.3: RISC-V Instruction Encodings

right-hand-side are suggestive of the class of instructions that use that coding:

- R: "register class": typically two register inputs
- I: "immediate class": typically one register and one immediate input
- S: "store class": for the STORE instructions SB, SH and SW
- B: "branch class": for the conditional branch instructions
- U: "Upper Immediate class": for LUI (Load Upper Immediate) and AUIPC (Add Upper Immediate to PC)
- J: "Jump class": for jump instructions JAL and JALR

We use standard Verilog "bit-slice" notation to refer to bit-fields of the 32-bit instruction. Thus, instr[6:0] refers to the lower 7 bits (bits 0 through 6) of the 32-bit instruction.

The overall "operation code" (opcode) of an instruction is a combination of 6-bit `opcode` field in instr[6:0] and the funct3 and funct7 fields at other positions in the instruction.

If the instruction reads at least one register, the first one is specified in the rs1 field ("register source 1"); the 5 bits in instr[15-19] specify one of the 31 registers. If the instruction reads two register, the second one is specified in the rs2 field; the 5 bits in instr[24-20] specify one of the 31 registers. If the instruction writes a register, it is specified in the rd field ("register destination"); the 5 bits in instr[11-7] specify one of the 31 registers.

In RISC-V, reading register `x0` always yields the value 0. This is a useful convenience built into the ISA. For example, there is an instruction to test if the rs1-value is less than the rs2-value. If we specify `x0` for rs2, then we effectively test if the rs1 value is negative.

Some instructions take input data from bits in the instruction itself; these are called "immediate" fields. In Figure 2.3 we can see that there are various encodings for the immediate fields. Consider the J-type instruction. Figure 2.4 shows how the 20 "imm" bits in the instruction are permuted, with a 0 bit appended to the right, to produce a 21-bit immediate value to be used in the computation.

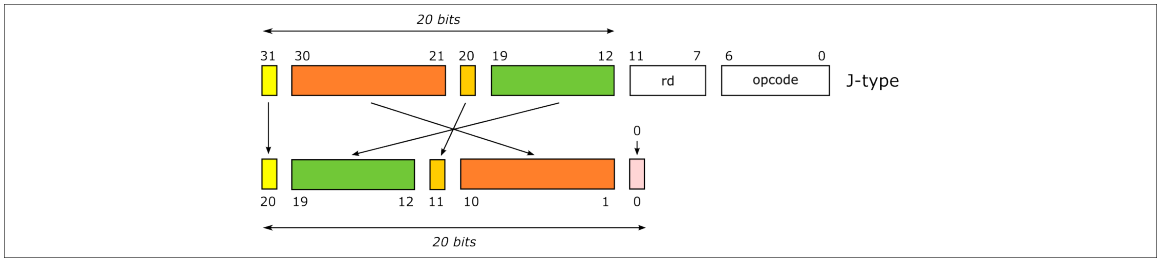



Figure 2.4: Construction of 21-bit immediate from 20 "imm" bits in J-type instructions

Similarly, Figure 2.5 shows how the 12 "imm" bits in the instruction are permuted, with a 0 bit appended to the right, to produce a 13-bit immediate value to be used in the computation.

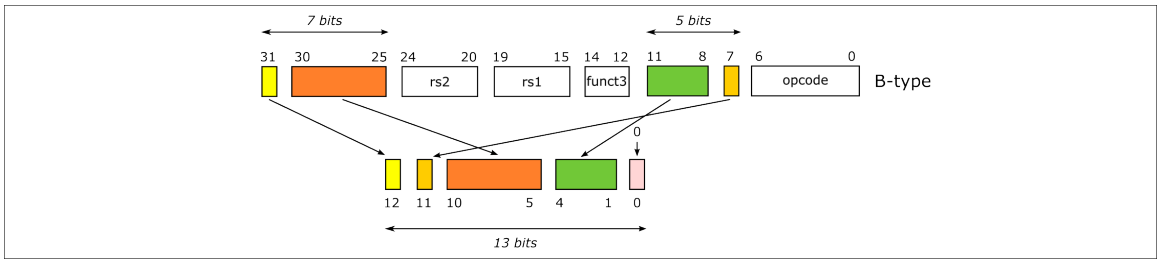



Figure 2.5: Construction of 13-bit immediate from 12 "imm" bits in B-type instructions

In both of the above, the encoding takes advantage of the fact that Jump and Branch target PCs are always at least 2-byte aligned, and therefore we do not have to use up an instruction "imm" bit to represent the "0" least-significant bit. The extra bit in the immediate value effectively doubles the "distance" that a jump or branch can span.[1]

The fact that there are so few encoding formats justifies the "RISC" in the ISA name: Reduced Instruction Set Computer. Having fewer formats, with few or zero special cases, simplifies the hardware needed to decode an instruction.

## 2.6 Unprivileged ISA RV32I

Figure 2.6 reproduces the table on page 130 of the RISC-V Unprivileged ISA specification, which shows all RV32I instructions. On the right-side of the diagram we have labeled different classes of instructions.

Note that there are just 40 instructions (again justifying the "RISC" name).

### 2.6.1 "Upper Immediate" instructions LUI and AUIPC

Figure 2.7 reproduces the top of page 19 of the RISC-V Unprivileged ISA specification, which describes the semantics of the LUI and AUIPC instructions in prose text.

[1] Actually in RV32I and RV64I PC targets are 4-byte aligned. The 2-byte alignment here accomodates the optional C (Compressed) extension, where instructions may be only 2-byte aligned.

**RV32I Base Instruction Set**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[31:12] | | | | rd | 0110111 | LUI |
| imm[31:12] | | | | rd | 0010111 | AUIPC |
| imm[20\|10:1\|11\|19:12] | | | | rd | 1101111 | JAL |
| imm[11:0] | | rs1 | 000 | rd | 1100111 | JALR |
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | | rs1 | 101 | rd | 0000011 | LHU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |
| fm \| pred \| succ | | rs1 | 000 | rd | 0001111 | FENCE |
| 000000000000 | | 00000 | 000 | 00000 | 1110011 | ECALL |
| 000000000001 | | 00000 | 000 | 00000 | 1110011 | EBREAK |

"load immediate"-kind: to load constant values into a register

"jump-and-link"-kind: subroutine calls and returns; distant jumps

"conditional branch"-kind: test and possibly jump up to ~0x1000 distance

"load data from memory into register" (rs1 and imm specify address)

"store data from register rs2 to memory" (rs1 and imm specify address)

"integer arithmetic operations (register-immediate)"

"integer arithmetic operations (register-register)"

"system" operations (ignore FENCE for now)

Figure 2.6: RISC-V RV32I Instructions



SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

| 31 12 | 11 7 | 6 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

Figure 2.7: RISC-V LUI and AUIPC Instruction semantics

Figure 2.8 is a pictorial depiction of the semantics of an LUI instruction. At the left of the

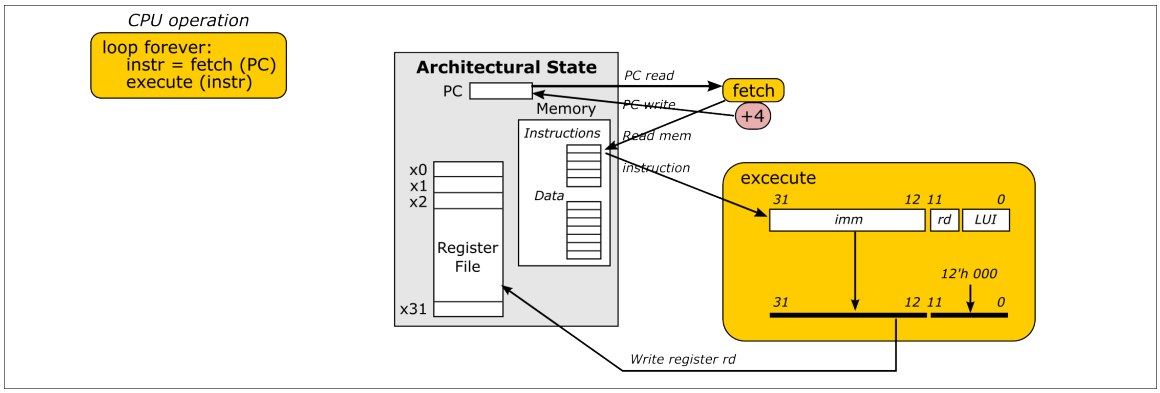



Figure 2.8: Execution semantics for LUI instructions

figure is depiction of the general behavior of a CPU, namely to loop forever, at each iteration fetching the instruction that is in memory at the address specified by the PC register, and then executing that instruction. For the LUI instruction, to the 20 bits [31:12] of the instruction (the "immediate" value) we append twelve bits of constant 0 as least-significant bits to construct a 32-bit value. This value is then written into register rd.

Figure 2.9 is a pictorial depiction of the semantics of an AUIPC instruction. As with LUI,

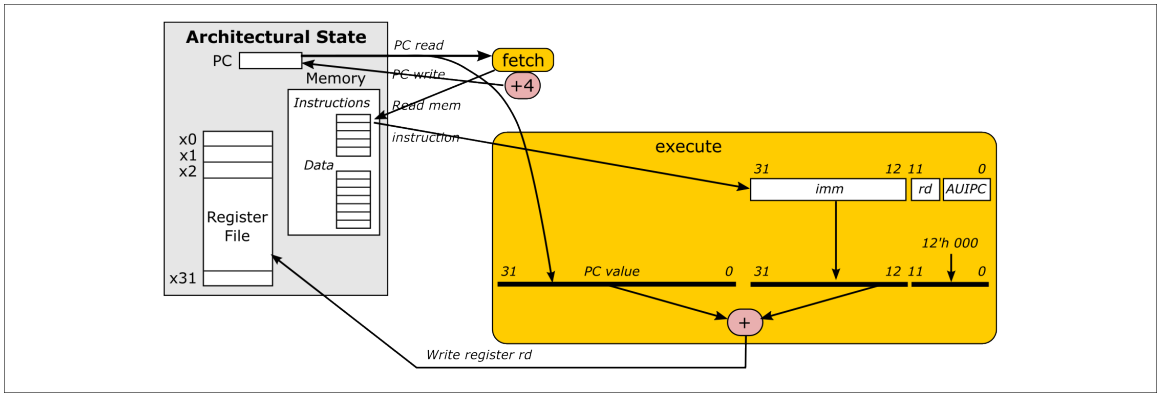



Figure 2.9: Execution semantics for AUIPC instructions

to the 20 bits [31:12] of the instruction (the "immediate" value) we append twelve bits of constant 0 as least-significant bits to construct a 32-bit value. For AUIPC we add this value to the 32-bits of the PC, and the result is then written into register rd.

### 2.6.2 Conditional BRANCH instructions

Conditional branch instructions compare the values in registers rs1 and rs2 for some condition: BEQ (equal), BNE (not-equal), BLT (less-than), BGE (greater-than-or-equal). If the condition is:

- false ("branch not taken"):
  - The instruction is a no-op, it just falls through to the next instruction (PC := PC+4).

- true ("branch taken"):
  - The 12 "imm" bits form a 13-bit value (see Figure 2.5).
  - This 13-bit value is sign-extended to 32 bits and added to the current PC to form a target address ("branch target").
  - The PC is updated to contain the target address.

Because of sign-extension (+/-), the branch may be forwards or backwards relative to the current PC.

For BLT and BGE, the comparison treats the two input 32-bit values as *signed* values. For BLTU and BGEU, they are treated as *unsigned* values.

### 2.6.3 LOAD and STORE memory-access instructions

LOAD (LB, LH, LW, LBU, LHU) instructions move data from memory into a register. STORE (SB, SH, SW) instructions move data from a register to memory. In both cases, the address is formed by sign-extending the 12-bit immediate value to 32 bits and adding it to the value from register rs1. For LOAD instructions, the value loaded is placed into register rd. For STORE instructions, the value in register rs2 is stored to memory, and there is no result value written into any register.

As with most modern ISAs, RISC-V memory is byte-addressed, *i.e.,* each address refers to a specific byte in memory. The size of the data to be loaded/stored is given by "B" (1 Byte), "H" (Halfword = 2 bytes), or "W" (Word = 4 bytes) in the instruction name.

The difference between LB and LBU is whether the 8 bits loaded (1 byte) are sign-extended or zero-extended, respectively, to 32 bits when placed in the 32-bit rd. (And similarly for LH vs. LHU.)

How to read the ISA spec: examples LOAD/STORE, Register-Register Arithmetic and Logic, Register-Immediate Arithmetic and Logic, Unconditional Jump.

### 2.6.4 Register-Register Arithmetic and Logic instructions

These instructions read registers rs1 and rs2, perform the specified arithmetic/logic operation on the two values, and store the result into register rd.

SLT ("set if less than") tests if register[rs1] < register[rs2], and writes 0 (false) or 1 (true) into the destination register, treating the inputs as 32-bit signed values. SLTU treats inputs as unsigned values.

SRL ("shift right logical") shifts the value from register[rs1] to the right (towards the least-significant bits) by the amount in register[rs2]. Zeroes are shifted in from the left.

In SRA ("shift right arithmetic"), the register[rs1][31] is shifted in from the left, *i.e.,* the 32-bit value is treated as a signed integer (bit 31 is the sign bit).

In SLL ("shift left logical"), zeroes are shifted in from the right.

In all three shifts, since there are only 32 bits in register[rs1], only the lower 5 bits of register[rs2] are relevant (*i.e.,* bits [4:0]); the rest are ignored.

### 2.6.5 Register-Immediate Arithmetic and Logic instructions

These instructions read register rs1, form a signed 32-bit value from the 12-bit "imm" bits, and perform the specified arithmetic/logic operation on the two values, and store the result into register rd.

In SRLI ("shift right logical immediate"), SRAI ("shift right arithmetic immediate"), and SLLI ("shift left logical immediate"), the 5-bit `shamt` field provides the "shift amount".

### 2.6.6 Unconditional Jump instructions

JAL ("jump and link") forms a 21-bit value from the 20 "imm" bits (see Figure 2.4), sign-extends it to 32 bits, adds it to the current PC value and updates the PC with the result.

JALR ("jump and link register") forms a 12-bit value from the 12 "imm" bits, sign-extends it to 32 bits, adds that to the value in register[rs1], forces bit [0] to be 0, and updates the PC to that value

Both JAL and JALR store the current PC + 4 ("return address") into register[rd].

These are most often used for subroutine calls and returns. Figure2.10 illustrates the protocol.

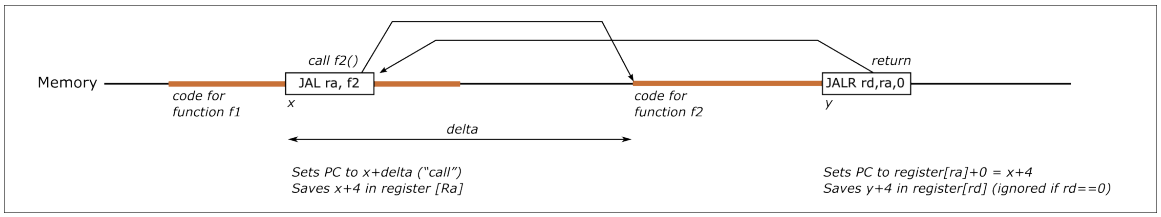



Figure 2.10: JAL and JALR for subroutine call and return

Suppose, inside function f1(), at address x, we have a `JAL ra,f2` instruction representing a subroutine call to function f2(). The compiler/linker will place "delta" in the JAL instruction's "immediate" field, where delta is the difference between x and the entry point of f2(). In RISC-V Assembly Language, "ra" ("return address") is another name for register[1], which is used to hold return addresses as part of the standard software calling convention. The JAL instruction saves x+4 (the return address) in register ra, and sets the PC to x+delta, so that the next instruction fetched will be the entry point of f2().

Inside f2(), at address y, suppose we have a `JALR 0,ra,0` instruction representing the return to the caller. It saves y+4 in register 0, but recall that writes to register 0 are always ignored. It reads the value in register ra, adds 0 to it and sets the PC to the result value, *i.e.,* the next instruction fetched will be from x+4.

JAL is used for most subroutine calls which are to a manifestly known subroutines (so the compiler/linker can compute the "delta" for the immediate value). JALR is used for subroutine calls where the "delta" is not known to the compiler/linker, for example when calling through a table of function pointers.

JALR is also used for subroutine calls and returns and conditional branches to "distant" target addresses. Remember that BRANCH instructions only have a 13-bit signed offset

from the current PC, and JAL only has a 21-bit signed offset. For more distant jumps, we can construct a full 32-bit value in a register (using one or more LUI and AUIPC instructions, for example) and then use that as the jump rs1 in a JALR instruction.

### 2.6.7 FENCE

The FENCE instruction is intended for RISC-V implementations that contain caches in the memory system. In such systems, data may not reach memory quickly or at all (is in the cache and is not evicted), and two items of data $x_1$ and $x_2$ may reach memory in a different order from the STORE instructions that wrote them (because the order in which their cache lines are written back to memory has no relation ship to the program STORE order).

The FENCE instruction is often used to "push" data from the cache to memory, although technically the FENCE instruction only guarantees an "ordering", *i.e.,* that if there are two STOREs $x_1$ and $x_2$ before and after a FENCE, respectively, then $x_1$ will be visible before $x_2$ to any other agent in the system (another CPU, a DMA engine, an I/O device, *etc.*).

## 2.7 Traps due to illegal instructions and other exceptions, and CSRs

Most processors cannot afford to get "stuck" on an unrecognized instruction (illegal instruction). An illegal instruction raises one of several possible "exceptions". Other events that raise exceptions are misaligned memory access (in FETCH or LOAD/STORE), or a memory-access to an unimplemented address (in FETCH or LOAD/STORE), *etc.*.

NOTE: "Illegal" just means: outside the currently implemented subset. For example (see Figure 2.2), a legal RISC-V instruction in, say, the M extension is considered illegal in an implementation that only implements the RV32I subset.

Exceptions are treated like an "unexpected call" to a special subroutine called a "trap handler". The architectural state is extended with a few extra registers belonging to a class of registers called CSRs (Control and Status Registers), shown in Figure 2.11.

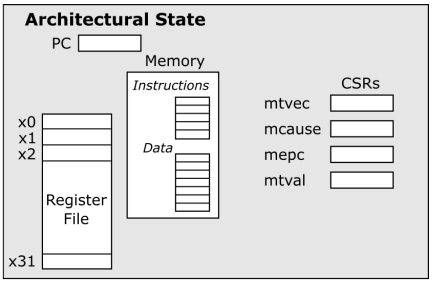



Figure 2.11: CSRs for handling traps

To "return" from a trap handler we need one more instruction: MRET, which is in the "RISC-V Privilege M ISA" (see Figure 2.2). Figure 2.12 illustrates the flow into the trap

handler and back on an exception (an ILLEGAL instruction is one possible cause of the exception).

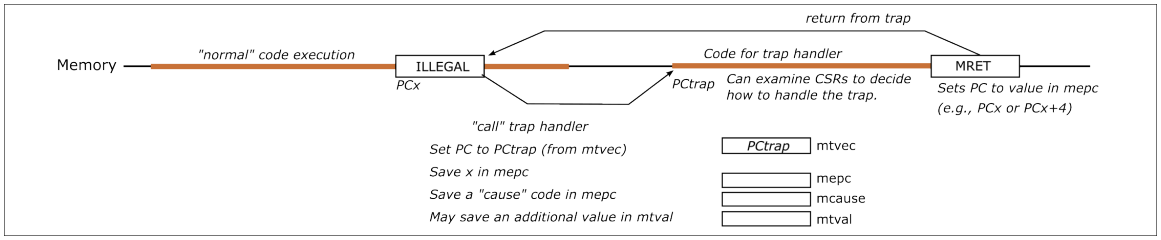



Figure 2.12: Trap and return flow

When the exception is detected, the hardware saves the faulting PC (PCx in the figure) in CSR MEPC, saves a cause-code in CSR MCAUSE and possibly one more piece of data (depending on the type of exception) in CSR MTVAL. Then, it sets the PC to the value in MTVEC, so that we start executing the trap-handler code.

When the trap-hanlder has finished, it executes an MRET instruction which copies the value in MEPC into the PC, continuing execution at that location.

Exception-cause codes relevant to this book (RV32I + exception handling) are shown in the table below (excerpted from Table 3.6 in the Privileged ISA Specfication document).

| Exception-Cause code | Description |
|---|---|
| 0 | Instruction address misaligned |
| 1 | Instruction access fault |
| 2 | Illegal instruction |
| 3 | Breakpoint |
| 4 | Load address misaligned |
| 5 | Load access fault |
| 6 | Store/AMO address misaligned |
| 7 | Store/AMO access fault |
| ... | ... |
| 11 | Environment call M-mode |
| ... | ... |

The trap handler code can examine MCAUSE, MEPC and MTVAL to determine what it should do to handle the trap. Regarding MEPC, it may:

- Leave MEPC untouched, so that, on MRET, we retry the exceptional instruction.

  *Example*: the exception was a page-fault due to a FETCH on an umapped page. The trap handler may map the page, and MRET to the faulting instruction to be retried (which, hopefully, should not page-fault again).

- Increment MEPC by 4, so that, on MRET, we resume at the next instruction after the faulting instruction.

  *Example*: the faulting instruction is a legal RISC-V instruction, but has deliberatly been left unimplemented (*e.g.,* to save hardware cost). The trap handler "emulates"

the instruction (*i.e.,* performs the required computation using implemented instructions), and resumes the normal flow at the next instruction.

*Example*: the trap handler just records that this instruction is illegal so that it can avoid it in subsequent execution.

- Change it to something else entirely, to abandon the current "normal" flow and do something else.

  *Example*: in an Operating System, we save MEPC for future resumption, and change it to give another process a chance to execute.

### 2.7.1 ECALL and EBREAK instructions, and Interrupts

RV32I instructions ECALL and EBREAK are handled just like other exceptions; in that sense, these are not "unexpected" conditions, but exceptions deliberately induced by the program. The only notable feature is that they place certain exception-cause codes in the MCAUSE CSR (see "Environment call M-mode" and "Breakpoint", respectively, in the table above).

In systems with operating systems, ECALL is used to move in a disciplined way from lower to higher privilege levels (from User mode to Supervisor mode, and from Supervisor Mode to Machine mode).

Some RISC-V implementations contain a hardware "Debug Module", in which case EBREAK is treated differently (please see the RISC-V Debug Module specification document). It is used as part of the implementation of the "break" command in debuggers like GDB, LLDB and OpenOCD.

An external interrupt is also dispatched into the trap-handler just like exceptions, the only difference being the code placed in the MCAUSE register (see Table 3.6 in the Privileged ISA Specfication document for all the defined interrupt cause codes).

### 2.7.2 CSRRxx instructions

In Section 2.7 we described how various CSRs are read and written during trap-handling. These reads and writes are performed *by the hardware* as part of taking a trap or performing an MRET.

But we also need a way to read and write CSRs *programmatically*, *i.e.,* from instructions in the program. For example, before any trap is taken, some early part of the execution needs to write the trap-vector's PC into MTVEC. During trap-handling, the trap-handler needs to read (and possibly write) MEPC, MCAUSE and MTVAL.

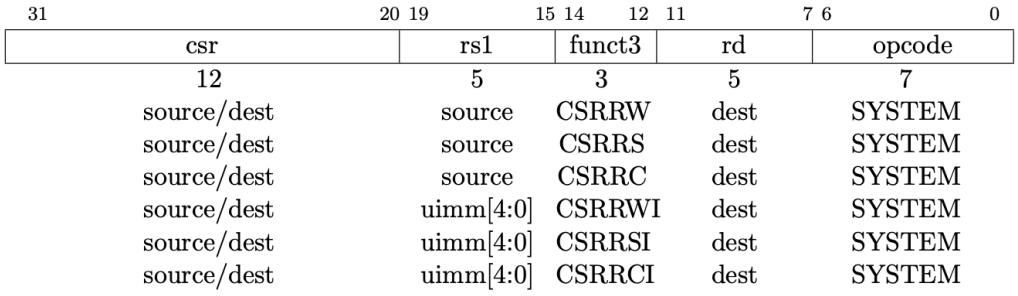
Programmatic access to CSRs is provided by a family of six "CSRRxx" instructions. The following table is excerpted from the RISC-V Unprivileged ISA spec document (Chapter 9, on the "Zicsr" extension).

As the text says, each CSR instruction:

- reads a value $x$ from a general-purpose register (rs1) or takes the rs1 field itself as a literal 5-bit value;

## 9.1 CSR Instructions

All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit *csr* field of the instruction held in bits 31–20. The immediate forms use a 5-bit zero-extended immediate encoded in the *rs1* field.



| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| csr | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| source/dest | source | CSRRW | dest | SYSTEM |
| source/dest | source | CSRRS | dest | SYSTEM |
| source/dest | source | CSRRC | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRWI | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRSI | dest | SYSTEM |
| source/dest | uimm[4:0] | CSRRCI | dest | SYSTEM |

Figure 2.13: CSRRxx instructions (from Unprivileged Spec)

- reads a value $y$ from a CSR (whose 12-bit address is given in `instr[31:20]`);
- returns $y$ into a general-purpose register (rd);
- and writes back a value into the CSR that is some function of $x$ and $y$ (the details vary across the six CSRRxx instructions).

For details, please read the RISV-V Unprivileged ISA specification document, Chapter 9.

## 2.8 RV64I differences from RV32I

The Architectural State for RV64I is just like that for RV32I, except that the PC and 32 registers are now 64-bits wide instead of 32-bits wide. Figure 2.14 reproduces the table of RV64I instructions from page 131 of the Unprivileged Spec document. RV64I starts with the same forty instructions as in RV32I, except that it replaces 3 of them—SLLI, SRLI and SRAI—with slight modifications: the "shamt" field is now 6 bits instead of 5, to accommodate 64-bit shifts.

In RV64I, the LW instruction loads 32-bits from memory, sign-extends it to 64 bits and stores it in the destination register. RV64I adds the LWU instruction that does the same, except that it zero-extends the 32-bit value from memory. RV64I also adds the LD instruction to load 64-bits from memory into a register.

RV64I adds the SD instruction to store a 64-bit value from a register into memory.

In RV64I, ADDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SRL and SRA all operate on 64-bit values. RV64I adds ADDIW, SLLIW, SRLIW, SRAIW, ADDW, SUBW, SLLW, SRLW and SRAW to operate on the lower 32-bits of 64-bit register values.

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |

**RV64I Base Instruction Set (in addition to RV32I)**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:0] | | rs1 | 110 | rd | 0000011 | LWU |
| imm[11:0] | | rs1 | 011 | rd | 0000011 | LD |
| imm[11:5] | rs2 | rs1 | 011 | imm[4:0] | 0100011 | SD |
| 000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 010000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |
| imm[11:0] | | rs1 | 000 | rd | 0011011 | ADDIW |
| 0000000 | shamt | rs1 | 001 | rd | 0011011 | SLLIW |
| 0000000 | shamt | rs1 | 101 | rd | 0011011 | SRLIW |
| 0100000 | shamt | rs1 | 101 | rd | 0011011 | SRAIW |
| 0000000 | rs2 | rs1 | 000 | rd | 0111011 | ADDW |
| 0100000 | rs2 | rs1 | 000 | rd | 0111011 | SUBW |
| 0000000 | rs2 | rs1 | 001 | rd | 0111011 | SLLW |
| 0000000 | rs2 | rs1 | 101 | rd | 0111011 | SRLW |
| 0100000 | rs2 | rs1 | 101 | rd | 0111011 | SRAW |

Figure 2.14: RV64I instructions (in addition to RV32I)

## 2.9 Continued Evolution of the RISC-V ISA (with your contribution?)

The RISC-V ISA is not frozen. As shown in Figure 2.2, the ISA has consciously been designed in a modular way and to be modularly extensible. RISC-V International (RVI, https://riscv.org) runs an organized process for the continued maintenance and evolution of the ISA. Special-interest groups drawn widely from RVI membership constitute various committees under the aegis of RVI to propose, develop, and specify new extensions to the ISA (for cryptography, for image and video manipulation, for high-performance computing, for AI, and so on). There is a formal public-review and ratification process for any new proposed extensions.

RVI has the usual corporate membership tiers seen in many consortiums but, unusually, it also has inexpensive memberships for individuals (students, hobbyists, ...) and academic institutions. So please feel free to join, in order to monitor the RISC-V ecosystem closely or, even better, to actively contribute to its future,

# Chapter 3

# RISC-V interpreters: the Design Space from Software Functional Simulators to High-Performance Hardware

Any artefact/engine that executes the instructions of any ISA is an *interpreter* for that ISA. The classical meaning of an interpreter is an algorithm (program) that examines/traverses a data structure that is itself the represention of a target program, and performs actions accordingly. In our case, the target program is a RISC-V binary and the data structure is an array or RISC-V instructions. The algorithm examines RISC-V instructions in the array, conceptually one-instruction-at-a-time, and performs the instruction's actions.

Any algorithm can be implemented in software or in hardware. Further, the boundary is fluid: parts of the algorithm can be implemented in software, cooperating with other parts that are implemented in hardware ("accelerators"). The choice between software and hardware implementation is pragmatic (speed, power, cost, cost of debugging and modification, cost of redesign, *etc.*); functionally there is no theoretical difference.

When we implement an ISA interpreter in software, we call it a "simulator". When we implement it in hardware, we call it a hardware implementation. Both software simulators and hardware implementations can vary widely in microarchitecture. Some design options are:

- Sequential or pipelined? One full instruction at-a-time, or multiple instructions flowing through a pipe, each at a more advanced step in its execution than the one behind it.
- Predictive (in pipelined implementations)? *E.g.,* predict what instructions to fetch while a BRANCH/JUMP flows through the pipe before we know the actual next-instruction determined the BRANCH/JUMP.
- Superscalar/VLIW? Fetch and execute more than one instruction in parallel, taking care to preserve sequential ISA semantics.
- Out-of-order? Execute each instruction as soon as its input data is available, without waiting for prior instructions which may still be waiting for their inputs.

For the same microarchitecture, a software simulator is typically *much slower* than a hardware implementation. This is because it involves (at least) two layers of simulation. The software simulator is itself a program that is being interpreted, perhaps directly in hardware. That program (the simulator), in turn, is interpreting the target ISA. The two interpreters need not and may not be for the same ISA. For example, if we run a RISC-V software simulator on a modern server, the lower level may be an x86 or ARM interpreter (*i.e.,* the CPU in in the server). A software simulator written in Python or Java involves three layers of ISAs, *e.g.,* hardware x86/ARM interpreting x86/ARM instructions representing a program to interpret bytecode (second level ISA), which, in turn represents an interpreter for RISC-V programs. Every additional layer of interpretation can slow down overall performance by possibly orders of magnitude.

Paradoxically, adding any of the microarchitectural details mentioned in the list above will normally slow down a software simulator but speed up a hardware implementation. This is because those microarchitetural details expose more *parallelism* and *concurrency* in the interpretation algorithm. Hardware implementations actually execute these parallel actions in parallel, whereas a software simulator (written, say, in C/C++) may execute them sequentially (*i.e., modeling* parallelism but in fact being sequential). Of course, the extra hardware speed is not free: it needs more hardware and more complexity in the design (cost, power consumption).

## 3.1 The RISC-V designs in this book

In this book we will focus on two simple hardware implementations, Both designs are coded in BSV, a free, open-source, modern, High-Level Hardware Design Language (HLHDL). BSV code can be compiled into Verilog, which can then be run on any Verilog simulator, or can be further processed by FPGA tools to run on FPGAs, or by ASIC tools for ASIC implemenetations. For more discussion of our choice of BSV, please see Appendix B.

Our first hardware RISC-V implementation—"Drum"—will be a simple one-full-instruction-at-a-time interpereter, almost a direct transliteration into BSV code of the generic ISA execution algorithm to be described next in Section 3.2. It does not implement any interesting microarchitectural feature, not even pipelining, which is the most basic microarchitectural feature of most CPU implementations. Lacking microarchitectural features, in fact the BSV code will look very similar to what you might write in C/C++ for a purely functional RISC-V simulator. Being written in BSV, however, we can compile and run it on actual hardware (FPGAs, ASICs).

Drum will not be fast compared to other hardware CPUs, because of lack of microarchitectural features, but we should still be able to run it at several 100 MHz on an FPGA, which will make it faster than many software functional simulators. It will be small (silicon area, and therefore low power as well). Drum is covered from Chapter 6 through Chapter 10.

Our second implementation—"Fife"—adds pipelining. Pipelining introduces new complications because of potential interaction between instructions that are at different stages in the pipe. We can focus on these new complications because all the functional aspects of RISC-V ISA execution have already been addressed in Drum. In fact, we will reuse the functional code from Drum without change. Fife is covered in Chapter 12 through Chapter 13.

For both Drum and Fife, we will focus initially on only the RV32I option of the RISC-V ISA. Please refer to the specification document "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA" [15]. In particular, look at Chapter 24 "RV32/64G Instruction Set Listings", and the first table therein, entitled "RV32I Base Instruction Set", showing forty instructions. These instructions are describe in more detail in the same document in Chapter 2 "RV32I Base Integer Instruction Set, Version 2.1".

We will extend this with just enough functionality to be able to recover from illegal instructions (*i.e.,* an instruction outside the set of forty RV32I instructions) and to handle interrupts. This minimal functionality will be taken from the specficitation document "The RISC-V Instruction Set Manual Volume II: Privileged Architecture"[16].

Beyond this book, we extend Drum and Fife to handle RV64I and more Unprivileged ISA options—M: integer multiply/divide, A: atomics, FD: single-and double-precision floating point, and C: compressed. We also handle more privileged ISA options—Privilege levels (M: Machine, S: Supervisor and U:User; full complement of Control and Status Registers (CSRs); Virtual Memory). With these extensions, Drum and Fife will be able to a full-feature Operating System (OS), such as Linux.

## 3.2 Abstract algorithm for interpreting an ISA

From our previous study of the RISC-V ISA, we know that the basic integer "architectural state" of a RISC-V CPU is very simple:

- A "program counter" (PC) indicating the address in memory of the next instruction to be executed.
- A "register file" consisting of 32 general purpose registers (GPRs), each containing data.

The PC and each register are either 32-bits wide (in the RV32 option of RISC-V) or 64-bits wide (in the RV64 option). For simplicity, we'll focus on RV32 here, but everything we discuss also applies to RV64.

Interpreting a program involves the repetition of a few simple steps,[1] illustrated in Figure 3.1:

- The "Fetch" step reads the current value of the PC and uses that value as an address in memory from which to read an instruction. Then, we proceed to the "Decode" step.
- The "Decode" step examines the fetched instruction to check if it is legal, to classify its major category (such as Control, Integer Arithmetic/Logic, or Memory), and to extract some properties such as which GPRs it reads (if any) and which GPR it writes (if any). Then, we proceed to the "Register-Read and Dispatch" step.
- The "Register-Read and Dispatch" step reads the GPRs for the instruction's inputs. Then, we proceed to one of the "Execute" steps, based on the category of the opcode in the instruction (Branch/Jump, Integer Arithmetic/Logic, or Memory).

[1] We prefer the word "step" here instead of "stage", which we will reserve to refer to stages in a hardware pipeline such as Fife.

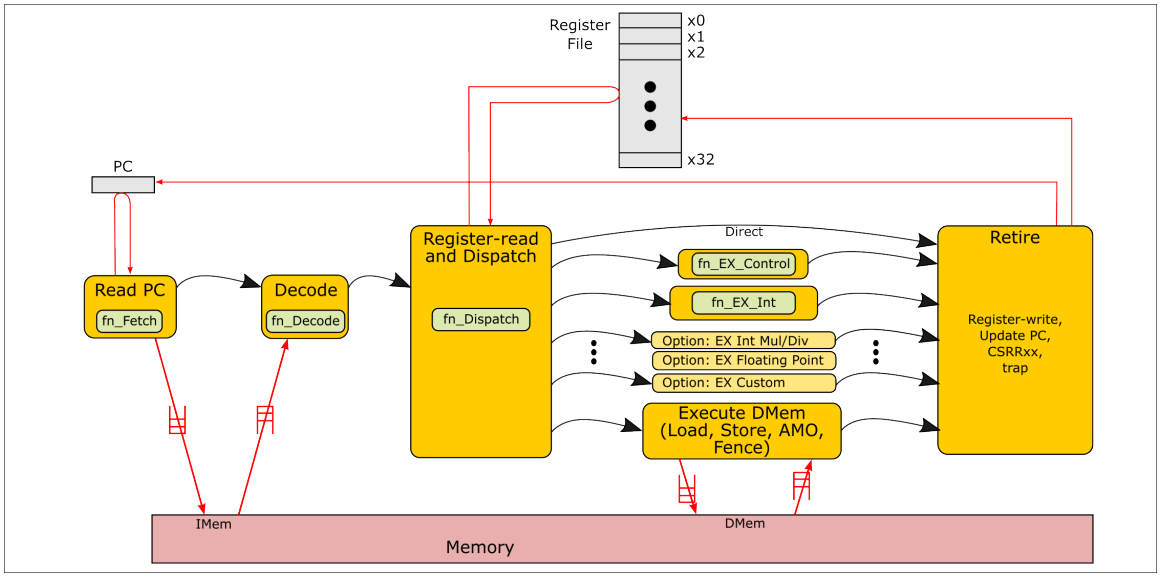



Figure 3.1: Simple interpretation of RISC-V instructions

- The "Execute Control" step is used for conditional-branch and jump instructions. For the former it evaluates the branch condition and, if true, and updates the PC to the branch-target PC. For jump instructions it updates the PC to the jump-target PC. Then, it goes back to the Fetch step to interpret the next instruction.

- The "Execute Integer Arithmetic and Logic" step is used for integer arithmetic and logic operations (addition, subtraction, boolean ops, shifts, *etc.*). Then, we proceed to the "Register-Write and Dispatch" step.

- The "Execute Memory Ops" step calculates a memory address based on an input value (that was read from a GPR) and reads or writes memory at that address. Then, we proceed to the "Register-Write and Increment PC" step.

- The "Register-Write and Increment PC" step writes the result from the previous Execute step back into a GPR, and increments the PC. Then, it goes back to the Fetch step to interpret the next instruction.

Thus we repeat these steps forever, instruction after instruction, starting each time at the Fetch step.

### 3.2.1 Memory latency and split-phase memory transactions

In Figure 3.1, note that the memory accesses, Fetch⟶Memory⟶Decode and Execute DMem⟶Memory⟶Execute DMem, are each shown with two arrows, one going to and another returning from memory. This is because, in any computer system, memory access is never "instantaneous". A request is sent to memory in one "clock tick" and the response is returned no earlier than the next clock tick. We also say: memory-access always has some *latency*.

In fact it may take several clock ticks, and a varying number of clock ticks. Memory accesses can go through *cache* systems, which can return data quickly on a "hit" and take several cycles on a "miss". Memory accesses can go to locations in I/O devices, which may be much "further away" in the memory subsystem, again possibly taking many cycles. Further, whether it takes one tick or several is not predictable—it can depend on the address of the request, and on past history of accesses which may leave caches in various states. It can also depend on technology replacement: next year's memory system may be faster than today's.

Thus,

- it is best to think of the path to memory (requests) and back (responses) as a *pipeline* or *queue* (FIFO) of unspecified length. The arrows in Figure 3.1 are decorated with small FIFO icons to emphasize this view.[2]
- Our CPU should not depend on any particular memory latency, *i.e.,* it should be robust to changes and variations in latency.

We also say that all memory transactions are *split-phase*: a request, followed later by a response (perhaps significantly later).

## 3.3 Plan for the order in which we tackle topics

This book serves two concurrent purposes: learning how to implement the RISC-V ISA and, specifically, how to implement it by coding it in BSV ("BSV learning"). The order in which we tackle topics is guided by the BSV-learning purpose, not by the step-by-step organization of Figure 3.1.

At the center of each step in Figure 3.1 are pure functions to decide what kind of instruction each 32-bit instruction is, perform arithmetic instructions, calculate addresses in memory, calculate conditions on whether to branch or not, etc. These pure functions are "combinational" functions, which we tackle in the next couple of chapters.

Note, we are *not* going to descend to the level of simple logic gates, how to optimize them, or how to implement higher-level combinational functions such as adders and multiplexers in terms of gates. These activities are today routinely handled by excellent compilers ("synthesis tools"). Our lowest-level combinational circuits, the ones we take as primitives, will be so-called "RTL-level" operators for arithmetic, shifts and logic operators on bit-vectors (`+`, `-`, `<<`, `>>`, `&&`, `||`, `^`, `!`, verb| |, and so on).

[2] Advancd memory systems may not even be FIFO-like; the CPU may receive responses in a different order from the original requests, in the interests of returning a response as soon the data is available, rather than waiting for its turn. In this book we will assume FIFO-like, in-order memory-accesses.

# Chapter 4

# BSV: Combinational circuits for the RISC-V step functions

## 4.1 Introduction

It is useful to start with the Decode step of Figure 3.1 because it involves bit-vectors, operations on bit-vectors, conditionals to classify instructions into classes, and `enum` types to name and encode instruction classes.

The inputs to the Decode step as depicted in Figure 3.1 are:

- A 32-bit piece of data—a RISC-V instruction—that has become available by reading it from memory at the PC address.[1]
- Any additional information passed on from the Fetch step.

The outputs of the Decode step have information needed by the next step (Register-Read and Dispatch). For a RISC-V instruction, useful information includes:

- Was the Fetch itself successful, or did it encounter a memory error; if so, what kind of memory error?
- Is it a legal 32-bit instruction?
- If legal, what is its broad classification: Control (Branch or Jump)? Integer Arithmetic or Logic? Memory Access? This will help in choosing the next step to which we must dispatch to execute the instruction.
- Does it have zero, one or two input registers? If so, which ones? This will help the next step in reading registers.
- Does it have zero or one output registers? If so, which one? This will help the final Register Write step in writing back a value to a register.

To compute these values, we need to examine "slices" of the 32-bit instruction ("bit vector"), such as the 7-bit "opcode" slice, the 5-bit "rs1", "rs2" and "rd" slices, and so on. We need to be able to compare these slices to constants (*e.g.,* "Is the opcode a BRANCH opcode?"). We need to do things conditionally, *e.g.,* if it is a BRANCH instruction, then it has an rs1 and rs2 slice but no rd slice, but if it is a JAL instruction it has an rd slice but no rs1 or

[1] When implementing the so-called "C" RISC-V ISA extension ("compressed instructions"), instructions can also be 16-bits, but we ignore that for now.

rs2. Finally, as in all good programming languages, we'd need to be able to package all this functionality inside a "function" with clearly specified input(s) and output(s). In the next several sections—4.2, 4.5, 4.6, 4.11, —we will learn the BSV concepts needed to code these ideas.

## 4.2 Bit Vectors

In BSV, as in many programming languages, every value has a *type*. The simplest, and lowest-level type in BSV is the bit-vector (a vector made up of a particular number bits). Later we will see that in BSV one can define more abstract types such as integers, booleans, vectors and arrays, lists, structs (records), tagged unions (algebraic types), trees, and so on. However, ultimately, any such value is represented in hardware as a bit-vector.

The BSV statement:

```
1   Bit #(32) pc_val;
```

declares the identifier `pc_val` to have the type `Bit#(32)`, *i.e.,* a bit-vector of 32 bits. The general syntax is similar to C or Verilog:

*type identifier*;

The BSC type `Bit#(32)` is roughly equivalent to the C type `uint32_t`. Unlike C, where only a few sizes are available, all multiples of 8 bits— (`uint8_t`, `uint16_t`, `uint32_t` and `uint64_t`)—bit-vectors in BSV can have any size (`Bit#(3)`, `Bit#(51)`, `Bit#(512)`, ...).

The bits in a BSV bit-vector of size $n$ are indexed from $n-1$ (most-significant bit) to 0 (least-significant bit). You can extract a *slice* of a bit-vector using usual Verilog notation:

```
1   Bit #(32) pc_val;
2   Bit #(12) page_offset = pc_val [11:0];
```

In the second line, we extract 12 bits of `pc_val` to get a bit-vector of size 12. BSV is *strongly typed* with respect to sizes, *i.e.,* it is very strict about matching sizes. For example, this statement:

```
1   Bit #(12) page_offset = pc_val [10:0];
```

will be reported as a type-error by the *bsc* compiler because the slice-expression on the right-hand side has type `Bit#(11)` which does not match the declared type `Bit#(12)`.

### 4.2.1 Built-in Operators on Bit Vectors

BSV bit-vectors can be compared for equality and inequality. BSV bit-vectors are synonymous with unsigned integers, and so a number of other operations are also available on bit-vectors. Examples:

```
Bit #(12) x, a, b, c, d, e, f;

// Comparison ops: result type is Bool
if (a == b) ...;        // equality
if (a != b) ...;        // not-equal to
if (a < b) ...;         // less-than
if (a <= b) ...;        // less-than-or-equal-to
if (a > b) ...;         // greather-than
if (a >= b) ...;        // greater-than-or-equal-to

// Arithmetic ops: result type is Bit #(12)
x = a + b - c * d;      // add, subtract, multiply

// Bitwise logic ops: result type is Bit #(12)
//   AND  OR   unary INVERT   XOR  XNOR  XNOR
x = a &  b |     (~ c)         ^   d ^~ e ~^ f;

// Shifts
x = (a << 3) & (b >> 14);    // left- and right-shift
```

Please see the *BSV Language Reference Guide* [1], Section 10.3, "Unary and binary operators" for a full list of available unary and binary operators. Unlike Haskell, in BSV you cannot define new unary or binary infix operators.

In such expressions, as usual bit-vector sizes must match exactly, else we'll get a type error, *e.g.,* we cannot compare a `Bit#(12)` value with `Bit#(11)` value. Unlike C and Verilog, BSV does not implicitly extend or truncate bit-vectors to match sizes.

Two functions are available to zero-extend and truncate bit-vectors.

```
Bit #(12) a;
Bit #(10) b;
b = a;                          // Type error: mismatched sizes
a = b;                          // Type error: mismatched sizes
b = truncate (a);               // Ok; truncates a to Bit #(10), then assigns
a = zeroExtend (b);             // Ok; extends b to Bit #(12), then assigns
if (a == zeroExtend (b)) ...    // Ok
if (truncate (a) < b) ...       // Ok
```

The functions `truncate()` and `zeroExtend()` are *polymorphic* in that they will truncate/extend by the appropriate amount as demanded by the context.

## 4.3 Integer types

BSV has four integer types, written as follows:

```
Bit #(n)        // bit-vectors, bounded to n bits
Int #(n)        // signed integers, bounded to n bits
UInt #(n)       // unsigned integers, bounded to n bits
Integer         // Mathematical integers (unbounded, no bit-width limit)
```

The most common type used in processor design (and possibly in all hardware design) is `Bit#(n)`. `UInt#(n)` can used to represent unsigned integers, $n$ bits wide. But since essentially the same operators (`+`, `-`, `&`, `|`, shifts, ..., see Section 4.2.1) are defined on both these types, this author mostly uses `Bit#(n)` for unsigned integers, and rarely uses `UInt#(n)`.

`Int#(n)` is used whenever we need to represent negative numbers and perform signed operations. These are represented in bits in hardware in "2's complement" representation (see https://en.wikipedia.org/wiki/Two%27s_complement).

`Integer` is used for true mathematical integers, *i.e.,* they are unbounded, ranging from minus infinity to plus infinity (in practice, limited by the amount of memory in your computer!). Being unbounded, they cannot be represented in any fixed-size hardware. `Integer` is used only for compile-time integers, where we do not need to fix a bound.

Fixed-width integer types all "wrap-around". For example, if we add 1 to a `Bit#(3)` value `3'b_111`, the result will be `3'b_000`; if we subtract 1 from a `Bit#(3)` value `3'b_000`, the result will be `3'b_111`.

Note that all four type names begin with an upper-case letter. This is true of all types in BSV: type names and enumeration constants begin with an upper-case letter, other identifiers begin with a lower-case letter.

## 4.4 Hexadecimal and Binary Notation for literal integers

BSV uses the same notation as Verilog and SystemVerilog for hexadecimal and binary literal integers. Some examples:

```
3'b010            // Binary literal, 3 bits wide
7'b_110_0011      // Binary literal, 7 bits wide
5'h3              // Hex literal, 5 bits wide
32'h3             // Hex literal, 5 bits wide
32'h_efff_0f17    // Hex literal, 32 bits wide (an AUIPC instruction)
'h23              // Hex literal, context determines width
```

As these examples show, a hexadecimal or binary integer literal is introduced by an optional bit-width, then a "tick" (single-quote) character, and then the binary or hexadecimal digits for the number. The character "`_`" may be used freely to space out groups of digits to improve readability for humans (the compiler ignores these spacers).

The last line shows that we can omit the size prefix, in which case the size will be inferred by the compiler from the context. For example, if we had:

```
   Bit #(32) pc_val = 'h_8000_0000;
```

then the literal is inferred to be 32 bits wide.

## 4.5 Boolean values

In BSV, `Bool` is the type of a Boolean value. It has the usual boolean operators `&&` (Boolean/logical AND), `||` (Boolean/logical OR) and `!` (Boolean/logical NOT).

### 4.5.1 Caution: `Bool` and `Bit#(1)` are different types

BSV is unlike languages like C and Python which are very loose about what can be used as a boolean value. For example in C, any non-zero numeric value or pointer is considered "True".

In BSV, `Bool` and `Bit#(1)` are *distinct* types, *i.e., bsc*'s type-checking will complain if one is used where the other is expected. This is because not all `Bit#(1)` values are meaningful as Boolean values.

The Boolean/logical operators mentioned above (such as `&&`) operate on `Bool` types and are distinct from the bit-wise logic operators mentioned earlier (such as `&`), which operate on `Bit#(n)` types.

Note that bitwise comparison operators, such as in the example `if (a <= b) ...` shown in Section 4.2 above, take `Bit#(n)` arguments and produce `Bool` results.

### 4.5.2 Example: recognizing legal RISC-V BRANCH instructions

The RISC-V ISA has a family of six conditional-branch instructions. Figure 4.1 is an excerpt from the Unprivileged ISA specification document [15]. The first line just gives us

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| imm[12\|10:5] | | rs2 | rs1 | funct3 | imm[4:1\|11] | opcode | B-type |
| imm[12\|10:5] | | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |

Figure 4.1: RISC-V conditional BRANCH instructions

the names of the various slices of a 32-bit BRANCH-type instruction, and the subsequent lines describe the six instructions. Note that they only differ in the `funct3` slice, where they use only six of the possible eight 3-bit codes.

Assuming `instr` is a 32-bit instruction, we can write BSV code to compute whether `instr` is or is not a legal BRANCH instruction:

```
1  Bit #(7) opcode_BRANCH = 7'b_110_0011;
2
3  Bit #(7) opcode = instr [6:0];
4  Bit #(3) funct3 = instr [14:12];
5  Bool legal = (opcode == opcode_BRANCH)
6               && (funct3 != 3'b010)
7               && (funct3 != 3'b011));
```

Line 1 defines `opcode_BRANCH` as a 7-bit constant whose binary value is 1100011. The `'7b` prefix indicates that the number should be read as a binary, not decimal, number. The "_"

underscore characters are present merely for our (human) readability, and have no semantic significance. Lines 3-4 extract relevant slices, and finally lines 5-7 define the desired legality condition.

Figure 4.2 shows the hardware circuit described by the code. Some observations:

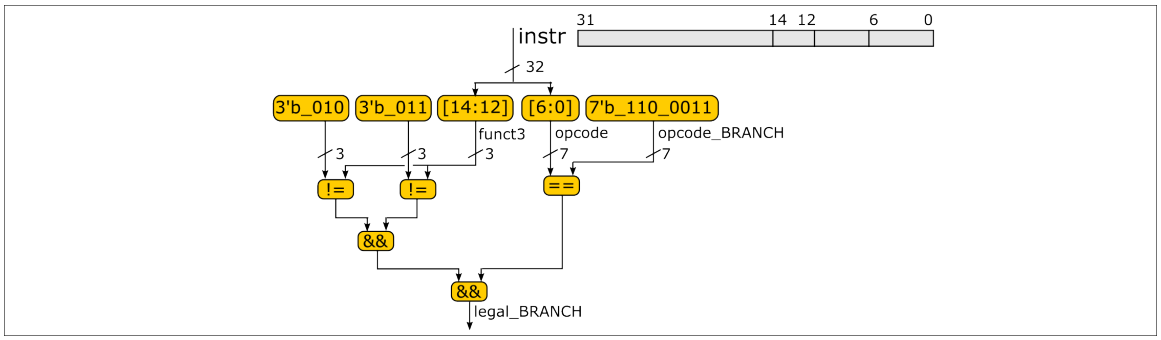



Figure 4.2: Testing for a legal BRANCH instruction

- Lines with arrow-heads in the figure represent bundles of one or more wires, also called "buses". For buses that have more than one wire, we show a small diagonal cross-hatch labeled with the number of wires (such as "3" or "7").
- Names/identifiers in BSV code that are bound to values are simply names for buses (in most software programming languages names represent memory locations; this is *not* the case in BSV).

### 4.5.3 Combinational circuits and primitives

Figure 4.2 is an example of a so-called *combinational* circuit. In general, a combinational circuit is any interconnection of combinational primitive "operators" that *does not contain cycles* (*i.e.,* a bus connecting back to an earlier part of the circuit). Examples of combinational primitive operators in BSV include comparisons (like `==` and `!=`), boolean operations (like `&&`), bit-slicing (`[n1:n2]`) truncation and extension, arithmetic (like `+`, `-`, `*`), shifts (`<<` and `>>`), and multiplexers (discussed in Section 4.11, later).

In BSV, and Verilog/SystgemVerilog RTL, we consider such operators as "primitive". In fact, such operators must themselves be implemented using lower-level circuit primitives such as AND, OR, and NOT gates which, in turn, must be implemented with even lower-level circuit structures such as transistors. We do not concern ourselves with such lower-level implementation because nowadays this is performed for us automatically by excellent so-called "synthesis" tools.

#### Combinational circuits have no side-effects (are "pure")

There is no "storage" in a combinational circuit, nor any concept of "updating" any storage (no "side-effects"). When a 32-bit value is presented at the input (top) of the circuit in Figure 4.2, conceptually we "instantly" see the 1-bit result at the output (bottom) of the circuit, *i.e.,* a combinational circuit is conceptually a pure, instantaneous, mathematical

function from intputs to outputs. If we change the 32-bit value presented at the input, conceptually the output changes instantaneously in response.

NOTE: Circuits are physical artefacts and must follow the laws of physics. Electrical signals will take some finite time to propagate from inputs to outputs through wires and silicon. This propagation delay will place a limit on the "clock speed" at which we are able to run a digital circuit. We ignore this for the moment, and discuss this in detail later.

## 4.6 Functions

The fragments of code shown above can be packaged into BSV functions, specifying precise types for argument(s) and result:

src_Common/Instr_Bits.bsv: line 31 ...

```
1 function Bit #(7) instr_opcode (Bit #(32) instr);
2    return instr [6:0];
3 endfunction
```

src_Common/Instr_Bits.bsv: line 35 ...

```
1 function Bit #(3) instr_funct3 (Bit #(32) instr);
2    return instr [14:12];
3 endfunction
```

src_Common/Instr_Bits.bsv: line 150 ...

```
1 function Bool is_legal_BRANCH (Bit #(32) instr);
2    let funct3 = instr_funct3 (instr);
3    return ((instr_opcode (instr) == opcode_BRANCH)
4            && (funct3 != 3'b010)
5            && (funct3 != 3'b011));
6 endfunction
```

Functions are invoked using the "application" syntax commonly used in most programming languages:

```
1    Bit #(32) x, y;
2
3    Bool result_x = is_legal_BRANCH (x);
4    Bool result_y = is_legal_BRANCH (y);
```

BSV function definition and application syntax is essentially the same as in SystemVerilog.

### 4.6.1 Pure functions vs. functions with side-effects (`Action`, `ActionValue`)

BSV has a system of data types and type-checking similar to Haskell in that it systematically distinguishes expressions which are "pure" (guaranteed not to have any side-effects) from expressions that may have side-effects.

The detailed reason for this distinction need not detain us now—suffice it to say here briefly that BSV's semantics are fundamentally based on a concept of "rules"; that rules

are condition-action pairs; that conditions *must not* have side effects (change the state of any hardware); and that the compiler needs to guarantee this, *i.e.,* that conditions are pure boolean expressions. These points will be discussed in more detail in Chapter 14.

One useful side-effect during debugging is the `$display()` statement. Why is `$display()` considered a side-effect? A pure expression does not modify any state, and therefore it can be optimized away by the compiler (evaluated zero times) if its result is never used. The compiler can also duplicate a pure expression (so it is evaluated more than once) for reasons such as cost (cheaper to recompute a value at some location in the code than to communicate the value that was computed elsewhere). Neither of these properties is true with `$display()`! Thus, the compiler needs to know about the purity or otherwise of every expression.

Keeping track of the purity or otherwise of an expression is not a local property. An expression may invoke a function which, in turn, invokes another function, and so on. The expression is pure only if there are no side effects anywhere in such a call chain (those functions may be defined in separate files, in libraries, and so on). BSV used a *monadic* type sytem (the same as in Haskell) to systematically track purity/impurity of expressions. Consider two function declarations:

```
1  function Bool f1 (...); ...
2
3  function ActionValue #(Bool) f2 (...); ...
```

Both of these functions return a boolean value. But an application `f1(x)` is *guaranteed* to be pure (by BSV's type system), whereas an application `f2(x)` is assumed possibly to have a side-effect. These functions also have different syntax for how they are invoked:

```
1  let result1 = f1 (x);
2
3  let result2 <- f2 (x);
```

The "`<-`" syntax is/can only be used to invoke functions with `ActionValue` type, and is also a good visual cue to indicate that the invocation is *performing some action* (side-effect) and also returns a value.

BSV also has an `Action` type, which is just a convenience for the special case of an expression that has a side effect and does not return any interesting value. You can think of `Action` as a synonym for `ActionValue #(void)`, where you can think of `void` as "uninteresting value". For example, you can think of `$display()` as a built-in function of type `Action`.

### 4.6.2 Combinational circuits = "doesn't have `Action` or `ActionValue` type"

Another pleasing consequence of BSV's type system is that we can identify precisely which expressions become combinational circuits. If the type of the expression is *not* `ActionValue#(`$t$`)` or `Action`, then it *must* be a combinational circuit, and *vice versa*.

### 4.6.3 Using `ActionValue` on pure functions for `$display` debugging

Sometimes when we write a complex pure function whose result type is $t$, we may deliberately write its result type as `ActionValue#(`$t$`)` so that we can insert `$display` statements for debugging inside the function body. If we merely inserted `$display` statements without changing the function type, the compiler will complain that the function does not type-check correctly.

We use this "trick" frequently in Drum and Fife source codes, for tracing and debugging. We will see many examples in the coming chapters.

## 4.7 A small testbench to test our code

Here is a small program to run our `is_legal_BRANCH()` function on a few tests:

```
1  import StmtFSM :: *;
2
3  function Bool is_legal_BRANCH (Bit #(32) instr);
4     ... as shown earlier ...
5  endfunction
6
7  (* synthesize *)
8  module mkTop (Empty);
9
10    mkAutoFSM (
11       seq
12          action
13             Bit #(32) instr_BEQ = {7'h0, 5'h9, 5'h8, 3'b000, 5'h3, 7'b_110_0011};
14             $display ("instr_BEQ %08h => %0d", instr_BEQ,
15                       is_legal_BRANCH (instr_BEQ));
16          endaction
17
18          action
19             Bit #(32) instr_BNE = {7'h0, 5'h9, 5'h8, 3'b001, 5'h3, 7'b_110_0011};
20             $display ("instr_BNE %08h => ", instr_BNE,
21                       fshow (is_legal_BRANCH (instr_BNE)));
22          endaction
23
24          action
25             Bit #(32) instr_ILL_op = {7'h0, 5'h9, 5'h8, 3'b100, 5'h3, 7'b_110_0000};
26             $display ("instr_ILL_op %08h => ", instr_ILL_op,
27                       fshow (is_legal_BRANCH (instr_ILL_op)));
28          endaction
29
30          action
31             Bit #(32) instr_ILL_f3 = {7'h0, 5'h9, 5'h8, 3'b010, 5'h3, 7'b_110_0011};
32             $display ("instr_ILL_f3 %08h => %0d", instr_ILL_f3,
33                       is_legal_BRANCH (instr_ILL_f3));
34          endaction
35       endseq);
36
37 endmodule
```

For the moment, don't try to understand all these boilerplate constructs in detail. Briefly, `mkAutoFSM` is like a sequential program (discussed in more detail in Section 10). It performs a sequence of four actions. In each action we define a 32-bit instruction with standard Verilog bit-concatenation syntax. For example, `instr_BEQ` is defined as a 32-bit value by concatenating a 7-bit hex 0 as an "immediate" value, a 5-bit hex 9 for rs2, a 5-bit hex 8 for rs1, a 3-bit 0 for funct3, a 5-bit hex 7 for rd, and a 7-bit binary value for the branch opcode. `instr_BEQ` and `instr_BNE` are legal branch instruction encodings. `instr_ILL_op` is not a legal branch instruction because it has the wrong 7-bit opcode in the opcode slice. `instr_ILL_f3` is not a legal branch instruction because it has an illegal 3-bit value in the funct3 slice.

In each action, the `$display()` prints the instruction in hex format, and prints the `Bool` result of applying `is_legal_Branch()` to the instruction. In two of the `$display()`s we print the `Bool` value as a decimal integer (`%0d` format). In the other two `$display()`s we use `fshow()` to print booleans as "`True`" or "`False`".

Suppose this code is in a file `Top.bsv`. We can now compile, link and execute the design (in simulation) as follows:

```
# ---- Compile BSV source code
$ bsc -u -sim Top.bsv
checking package dependencies
compiling Top.bsv
code generation for mkTop starts
Elaborated module file created: mkTop.ba
All packages are up to date.

# ---- Link to form a simulation exeutable
$ bsc  -sim  -e mkTop -o ./exe_HW_bsim
Bluesim object created: mkTop.{h,o}
Bluesim object created: model_mkTop.{h,o}
Simulation shared library created: exe_HW_bsim.so
Simulation executable created: ./exe_HW_bsim

# ---- Execute the simulator
$ ./exe_HW_bsim
instr_BEQ 009401e3 => 1
instr_BNE 009411e3 => True
instr_ILL_op 009441e0 => False
instr_ILL_f3 009421e3 => 0
```

---

**Exercise 4.1:**

Extend the testbench to test more 32-bit values with `is_legal_BRANCH()`.

**Exercise 4.2:**

Refer to the "RV32I Base Instruction Set" listing in "Chapter 24 RV32/64G Instruction Set Listings" in the RISC-V Unprivileged ISA specification document [15]. It lists 40 RV32I instructions. Similar to `is_legal_BRANCH()`, write BSV code for the following functions:

```
   function Bool is_legal_JAL (Bit #(32) instr);
      ... acccepts JAL

   function Bool is_legal_JALR (Bit #(32) instr);
      ... acccepts JALR

   function Bool is_legal_OP (Bit #(32) instr);
      ... acccepts LUI, AUIPC, ADD, SLT, OR, AND, ...

   function Bool is_legal_OP_IMM (Bit #(32) instr);
      ... acccepts ADDI, SLTI, ..., ORI, ANDI, ...

   function Bool is_legal_Mem (Bit #(32) instr);
      ... acepts LB, LH, LW, LBU, LHU, SB, SH, SW
```

Ignore FENCE, ECALL and EBREAK instructions; for the moment we'll treat them as illegal instructions.

**Exercise 4.3:**

Extend the testbench to test more 32-bit values with all the `is_legal_XXX()` functions.

□

---

## 4.8 `enum` types

In Figure 3.1, the Register-Read and Dispatch step needs to know which of the four alternative downstream paths should be selected for executing the insruction. In particular, we need to know whether the incoming instruction is a system instruction, a control instruction (branch or jump), an integer arithmetic/logic instruction, or a memory-accessing instruction. We could think of coding these "classes" using numbers (0 for system, 1 for control, 2 for integer, 3 for Mem), but it is more readable, and cleaner, to use an "enum" type (similar to enum types in SystemVerilog and C):

src_Common/Inter_Stage.bsv: line 40 ...

```
typedef enum {OPCLASS_SYSTEM,
              OPCLASS_CONTROL,    // BRANCH, JAL, JALR
              OPCLASS_INT,
              OPCLASS_MEM,        // LOAD, STORE, AMO
              OPCLASS_FENCE}      // FENCE

OpClass
deriving (Bits, Eq, FShow);
```

This defines a type `OpClass` containing five constants (the last two (`OPCLASS_MEM` and `OPCLASS_FENCE`) both indicate the memory-access path.

### 4.8.1 `deriving (Bits)`

Because we said "`deriving(Bits)`", the *bsc* compiler will automatically represent them with the obvious codes 0, 1, 2, 3 and 4 in a minimal bit-width (`Bit#(3)`). If we wanted an alternative (non-default) coding for these constants, we would *not* say "`deriving(Bits)`", and we would provide an explicit mapping function into codes (see "typeclass instances", later).

### 4.8.2 `deriving (Eq)`

Because we said "`deriving(Eq)`", the *bsc* compiler will automatically define the "equality" (and "inequality") functions for values of this new type, in the natural and obvious way (here, just compare the bit representations for equality). For other definitions of equality/inequality, we would *not* say "`deriving(Eq)`", and we would instead define equality/inequality explicitly (see "typeclass instances", later).

### 4.8.3 `deriving (FShow)`

Given a value *opclass* of type `OpClass`, if we directly print it, *e.g.,*:

```
1    opclass = OPCLASS_MEM;
2    $display ("opclass = %d", opclass);
```

the output will be "3", which is the numeric code the compiler assigns to the constant `OPCLASS_MEM` given its position in the list of labels in the `enum` definition shown in Section 4.8 (starting at zero).

```
1    opclass = OPCLASS_MEM;
2    $display ("opclass = %d", opclass);
```

Because we said "`deriving(FShow)`" in the `enum` declararion, the *bsc* compiler will automatically define an "`fshow()`" function for this type: if we print as follows:

```
1    opclass = OPCLASS_MEM;
2    $display ("opclass = ", fshow (opclass));
```

the output will be "OPCLASS_MEM", *i.e.,* the symbolic name.

## 4.9 Syntax of Identifiers

The syntax of an identifier (name) in BSV follows the same conventions as in many programming languages: any sequence of alphabets, digits and underscore characters, with the first letter always being an alphabet.

BSV follows the Haskell system where an identifier has a different roles depending on whether its first letter is lower-case or upper-case. An upper-case first-letter represents a *constant*, either a value constant or a type constant. All variables (value variables or type variables) begin with a lower-case letter.

In the enum type-definition in Section 4.8, the identifiers `OPCLASS_SYSTEM`, `OPCLASS_CONTROL`, `OPCLASS_INT` and `OPCLASS_MEM` are all value constants (they all begin with an upper-case letter). The identifier `OpClass` (and identifiers seen earlier: `Bool` and `Bit`) are all type constants. The identifiers `Bits`, `Eq`, and `FShow` are all typeclass constants.

Other variables seen earlier, like `x`, `y`, `a`, `b`, `opcode`, and `result_x` are all ordinary value variables.

## 4.10 Syntax of comments

Comments in BSV have the same syntactic conventions as in Verilog, SystemVerilog and C/C++:

- A pair of forward-slashes ("`//`") begins a comment that spans to the end of the current line.

  There are many examples of this in the code fragments already shown above.

- A region of text spanning multiple lines can be a comment if preceded by "`/*`" (forward-slash and asterisk) and followed by "`*/`" (asterisk and forward-slash).

  This form is often used to "comment-out" a region of text during debugging or trying out alternatives.

## 4.11 if-then-else statements and hardware multiplexers

In most programming languages, "if-then-else"‘ is a so-called "control" construct: depending on the boolean condition, either the then-arm or the else-arm is executed (*not both*!).

In BSV an "if-then-else‘ represents a hardware *multiplexer*. The then-arm and else-arm each represent hardware that computes some value. The if-then-else construct simply selects the output of one of the two arms and passes it on as its output. Stated another way, the if-then-else "multiplexes" the two arm-outputs into a single output. In programming-language terms, *both* arms of the conditional are always "executed"—each arm represents an actual piece of hardware that is continuously computing its output.

The data type of the condition in an if-then-else must always exactly be `Bool` (not `Bit#(1)`, not an integer, *etc.*). The types of the two arms of the conditional must be exactly the same, and this is also the type of the output of the output of the if-then-else.

For example, here is a function that distinguishes CONTROL instructions from integer instructions, returning an `OpClass` (Section 4.8):

```
1 function OpClass instr_opclass (Bit #(32) instr);
2    OpClass result;
```

```
   if (is_legal_BRANCH (instr)
      || is_legal_JAL (instr)
      || is_legal_JALR (instr))
      result = OPCLASS_CONTROL;
   else
      result = OPCLASS_INT;
   return result;
endfunction
```

This can also be written using so-called "conditional expressions" (using the same syntax as in SystemVerilog and C):

```
function OpClass instr_opclass (Bit #(32) instr);
   return ((is_legal_BRANCH (instr)
            || is_legal_JAL (instr)
            || is_legal_JALR (instr))
           ? OPCLASS_CONTROL
           : OPCLASS_INT);
endfunction
```

It's a matter of taste and style whether one uses if-then-else expressions or C-style conditional expressions. It may also depend on the size of the sub-epressions. The primary goal should be readability.

Both these code fragments represent the same hardware, shown in Figure 4.3. The 32-bit

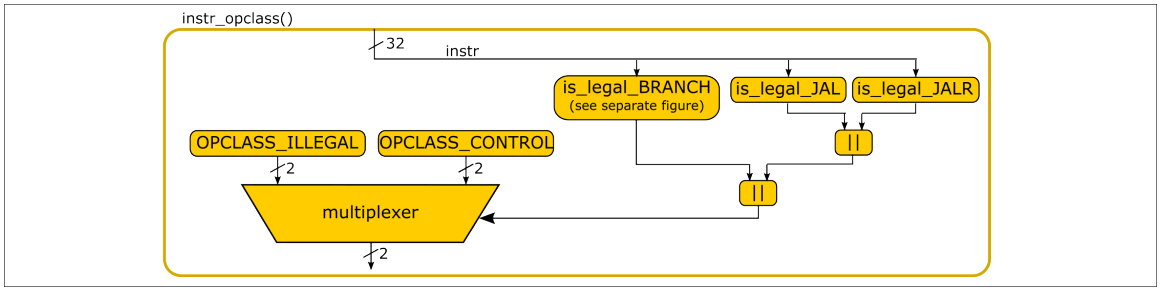



Figure 4.3: If-then-else is a multiplexer

`instr` argument is fed into the circuits for `is_legal_BRANCH()` (hardware schematic in Figure 4.2), `is_legal_JAL()` and `is_legal_JALR()` which are OR'd to produce a `Bool` output which, in turn, is used to select one of two 2-bit constant values, producing a final 2-bit result. The multiplexer, also called a "MUX" for short, is a primitive combinational circuit.

If-then-elses and conditional expressions can of course be nested:

```
function Bool instr_opclass (Bit #(32) instr);
   OpClass result;
   if (is_legal_BRANCH (instr)
      || is_legal_JAL (instr)
      || is_legal_JALR (instr))
```

```
6       result = OPCLASS_CONTROL;
7    else if (is_legal_OP (instr))
8       result = OPCLASS_INT;
9    else if (is_legal_OP_IMM (instr))
10      result = OPCLASS_INT;
11   else if (is_legal_Mem (instr))
12      result = OPCLASS_MEM;
13   ... and so on ...
14   return result;
15 endfunction
```

This represents a cascade of multiplexers in hardware, as shown in Figure 4.4

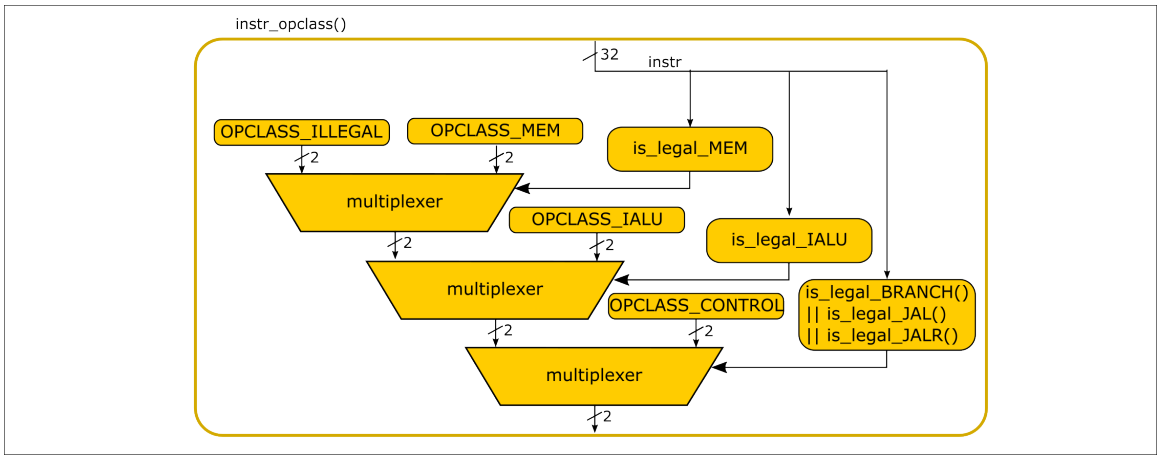



Figure 4.4: Nested if-then-elses become cascaded multiplexers

### 4.11.1 Parallel multiplexers and MUX synthesis

The circuit in Figure 4.4 has a serial structure—the `OPCLASS_CONTROL` branch has priority, and only if its condition is False can one of the other results flow through. Also observe that the longest path length increases *linearly* with number of classes—here, `OPCLASS_ILLEGAL` flows through all three multiplexers.

But we know from RISC-V instruction encodings that the `OPCLASS_CONTROL`, `OPCLASS_IALU` and `OPCLASS_MEM` conditions are *mutually exclusive*; no instruction simultaneously falls into more than one such class. In such situations (mutually exclusive conditions) it is possible to create more a efficient circuit called a *parallel MUX*. An exercise below shows how to create a parallel MUX explicitly, but in many cases downstream RTL-to-lower-level-hardware synthesis tools will do this automatically.

---

**Exercise 4.4:**

Write a testbench for the `instr_opclass()` function: pass in different 32-bit instructions to produce the op class, and print out the op class. When printing the class, try printing it as an integer, and also using `fshow()`.

**Exercise 4.5:**

Write a new version of the `instr_opclass()` function that expresses a parallel MUX instead of a priority MUX. The key ideas are:

- Define a value `x_CONTROL` that is either `OPCLASS_CONTROL`, or 0 (of the same bit-width) if the `Bool` values of `is_legal_BRANCH()`, `is_legal_JAL()` and `is_legal_JALR()` are all False. We can implement this by replicating the 1-bit Bool condition to the width of the `OpClass` type and bitwise-AND'ing this with `OPCLASS_CONTROL`.
- Similarly, define values `x_IALU` and `x_MEM` that is either `OPCLASS_IALU`/`OPCLASS_MEM` or 0 if the `Bool` value of `is_legal_IALU()`/`is_legal_MEM()` is False.
- Define a value `x_ILLEGAL` that is either `OPCLASS_ILLEGAL`, or 0 if any of the previous conditions is True.
- Finally, just bitwise-OR the four `x_XXX`values together to produce there result.

This kind of MUX is also called an AND-OR mux because of its structure. Note that it relies for correct operation on *precisely one* of the bitwise-OR arguments being True. Here, we are assured of this because of the mutual exclusivity of the conditions.

**Exercise 4.6:**

Sketch out a schematic diagram for the hardware of the parallel MUX in the previous exercise.

The schematic will clearly reveal the parallel nature of the MUX compared to the serial nature of the priority MUX shown earlier in the schematic in Figure 4.4.

How many multiplexers does each `OPCLASS_XXX` value flow through? How many bitwise-OR operators are needed? Note that we can organize the bitwise-OR operators as a balanced binary tree, so that the path through the circuit grows *logarithmically*, not linearly, with the number of classes.

**Exercise 4.7:**

Test your new version of the `instr_opclass()` function in your testbench.

□

---

## 4.12 Sharing code for RV32 and RV64 *via* parameterization

The RISC-V ISA is actually two ISAs—a 32-bit ISA called RV32 and a 64-bit ISA called RV64. These are not randomly different ISAs; they have been carefully engineered to overlap as much as possible:

- Most of the RV32 instructions are exactly the same in RV64
- Three R32 instructions are slightly different in RV64–the shift instructions SLLI, SRLI and SRAI have 5-bit shift-amounts in RV32 (allowing up to 32-bit shifts), whereas they have 6-bit shift-amounts in RV64 (allowing up to 64-bit shifts).

- RV64 adds several new instructions that compute on 64-bit values.

Because of this large overlap of RV32 and RV64, we would like to share BSV code as much as possible between RV32 and RV64, *i.e.,* we would like to parameterize our BSV code so that it can be re-used between RV32 and RV64 implementations.

### 4.12.1 Numeric types

We have mentioned the type `Bit#(n)` frequently so far, representing bit-vectors of width $n$ bits. All our examples showed a particular $n$, such as:

```
1  Bit #(32) instr;
2  Bit #(32) pc_val;
```

The first declaration is fine for both RV32 and RV64, since instructions are 32-bits wide in both. However, the second declaration only works in RV32, since the program counter is 64-bits wide in RV64 (type `Bit#(64)`).

The "32" or "64" argument to the `Bit#(n)` type is a *numeric type*. Although syntactically they look just like the *values* 32 and 64, when used inside a type-expression like `Bit#(n)`, they are not values, but numeric types. BSV's type-system carefully distinguishes between these two cases because numeric-types usually say something about *hardware structure*, which cannot be changed once created! So, while we can perform arbitrary arithmetic on numeric *values*, we cannot do so on *numeric types*.[2]

### 4.12.2 Type synonyms

In BSV we can define a new symbolic name for an existing type, and then we can use that symbolic names in place of the existing type. Example, from RV32 code:

```
1  typedef 32 XLEN;      // new name for numeric type 32
2
3  Bit #(XLEN) pc_val;
4  Bit #(XLEN) rs1_val;  // Value read from register rs1 in register file
5  Bit #(XLEN) rs2_val;  // Value read from register rs2 in register file
6  Bit #(XLEN) rd_val;   // Value written to register rd in register file
```

By changing the single definition in line 1 to:

```
1  typedef 64 XLEN;      // new name for numeric type 32
```

the remaining code will work for RV64 as well.

[2] A limited form of arithmetic is possible on numeric types. Consider a generic function that takes two arguments of type `Bit#(m)` and `Bit#(n)` and returns the concatenation of these bit-vectors: its output type is `Bit#(m+n)`. By limiting the available arithmetic *bsc* can resolve it completely "statically", *i.e.,* at compile time, before it even compiles to Verilog RTL. We ignore this for now, and discuss it later.

### 4.12.3 The numeric value corresponding to a numeric type

Although BSV keeps a strict separation of numeric types and numeric values (and limits the available arithmetic on the former), it is always safe to convert a numeric type into the corresponding numeric value, since these values are all known statically (at compile time). The built-in pseudo-function `valueOf()` is provided for this:

src_Common/Arch.bsv: line 26 ...

```
1 Integer xlen = valueOf (XLEN);
```

Here, `xlen` is an ordinary value variable whose integer value is the same as that expressed by the numeric type `XLEN`.

### 4.12.4 Conditional compilation

Just like in Verilog, SystemVerilog and C/C++, the *bsc* compiler runs BSV source code through a "pre-processor" before compilation, which can perform simple text ("macro") substitutions. Using this facility, we can pass an argument to the compiler that has the effect of configuring the source code for RV32 or RV64 (the following code is from Fife/Drum's `Arch.bsv` file:

src_Common/Arch.bsv: line 14 ...

```
1 `ifdef RV32
2
3 typedef 32 XLEN;
4
5 `elsif RV64
6
7 typedef 64 XLEN;
8
9 `endif
```

As in Verilog and SystemVerilog, pre-processor directives begin with a ` character (back-tick) (analogous to `#ifdef` in the C/C++ pre-processor).

When we invoke the *bsc* compiler, we can pass it command line arguments `-DRV32` or `-DRV64`; the pre-processor will then select the appropriate `typedef` line. Thus, we can write common code that will work for both RV32 and RV64. The integer value `xlen` will have the numeric value 32 or 64 depending on how it was compiled.

Pre-processor macros allow us to conditionally compile different source text based on the macro definitions we supply to the compiler. We can also compile alternatives based on the value `xlen`

```
1    if (xlen == 32) begin
2       ... code that must execute if we are in RV32 mode ...
3    end
4    else begin
5       ... code that must execute if we are in RV64 mode ...
6    end
```

Whenever possible, it is preferable to use the `if(xlen==...)` form instead of the `‘ifdef` form for conditional compilation because (a), the code is more readable and (b), as we experience in many languages, pre-processor macros can be quite dodgy (scoping, inadvertant variable capture, inadvertant surprises due to associativity of infix operators, ...).

Note that in the `if(xlen==...)` form both arms of the conditional must type-check correctly, whether `xlen` is 32 or 64. There are ways to achieve this with judicious use of bit-slicing, `extend()` and `truncate()`; we will point them out as we encounter them. If the two arms cannot both type-check whether `xlen` is 32 or 64, we may have to resort to the `‘ifdef` form.

There is zero run-time overhead in using the `if(xlen==...)` form because the *bsc* compiler will evalute the if-condition statically and reduce the if-then-else to just the relevant arm.

# Chapter 5

# BSV: Struct types, and RISC-V: Memory requests and responses

## 5.1 RISC-V: structs communicated between steps

Various kinds of information need to be communicated between the steps of Figure 3.1—program counter values, instructions, values read from registers, values to be written back to registers, and so on. `struct` data types (short for "structures") are suitable for bundling together heterogeneous collections of values. (This is the same concept in C and SystemVerilog; it is also called a "record" in some programming languages.) Each component of a struct is called a "field" or a "member" of the struct. Figure 5.1 annotates Figure 3.1 with struct

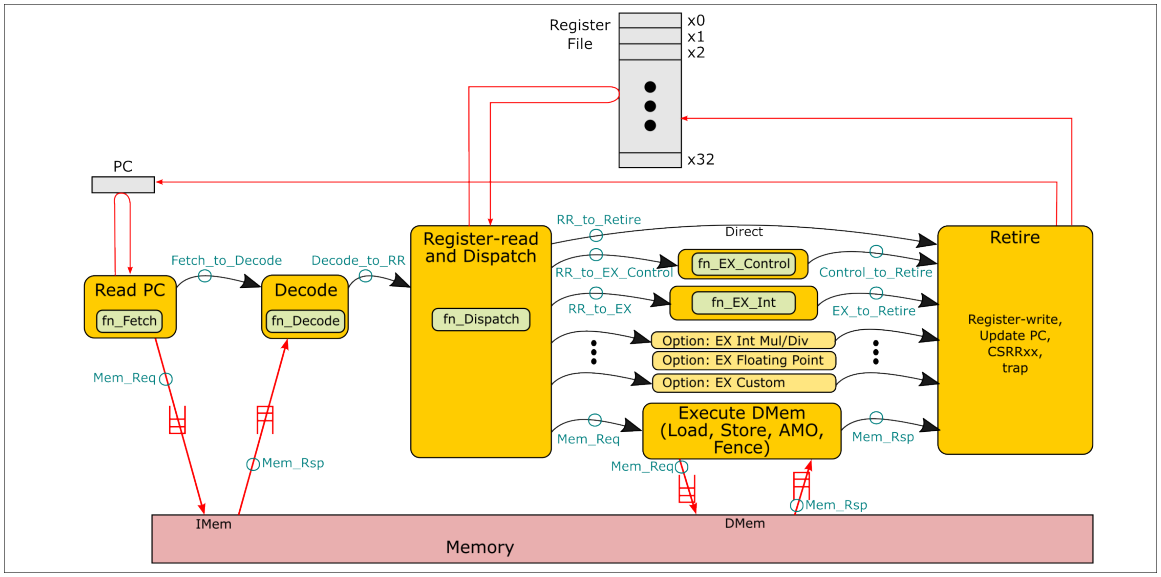



Figure 5.1: Simple interpretation of RISC-V instructions (Fig. 3.1 with arrows annotated with `struct` types)

types communicated on each of the black arrows between steps, and each of the red arrows

to and from memory. In this and the next few chapters we will flesh out the details of all these struct types. We will use exactly the same struct types for Fife and Drum, *i.e.,* whether the implementation is pipelined or not. All these `struct` declarations can be found in the file: `src_Common/Inter_Stage.bsv`.

## 5.2 BSV: `struct` types

Consider the black arrow from the Decode step to the Register-read-and-Dispatch step of Figure 5.1. We want to communicate several values, including:

- The current PC. This will be needed by BRANCH, JAL, JALR and AUIPC instructions to compute addresses that are offset from the current PC. It will be needed for any traps (exceptions) that may occur, which save PC for the trap-handler.
- An `exception` flag, indicating:
  - whether an error was encountered in the Fetch-to-Memory-to-Decode path, or
  - whether the Decode step's analysis indicates that the instruction is not legal (unrecognized 32-bit code).

  When the exception flag is true, a `cause` field provides more detail.
- If there is no exception (no Fetch memory error; instruction is legal), other fields provide more analytical detail for subsequent steps:
  - The "fall-through" PC, *i.e.,* the address of the next instruction following this one in memory. For RV32I and RV64I, this will always be PC+4, since all instructions are 4-bytes long.[1] For most instructions, the fall-through PC is indeed the unique next PC. For conditional BRANCH instructions, this is the next PC if the BRANCH is not taken. For JAL and JALR instructions (unconditional jumps), this is the "return address" saved by the instruction in a register.
  - The instruction itself. This will be needed for opcode details, the rs1, rs2 and rd register indexes, immediate values, *etc.*

  The next several fields are derived by analyzing the instruction. They could be re-derived wherever needed by re-analyzing the instruction, but we perform that work just once in the Decode step and communicate the results.
  - The `OpClass`. This indicates to which next-step in Figure 5.1 we dispatch for subsequent actions.
  - Whether the instruction reads rs1 and/or rs2 register values. This will be needed to control reading from the register file.
  - Whether the instruction writes an rd register value. This will be needed to control writing to the register file.
  - The "immediate" value in the instruction. Refer to the top of each page of "Table 24.2 Instruction listing for RISC-V" in the Unprivileged Spec [15], which shows that the I-, S-, B-, U- and J-type instructions have immediate values of different sizes and encode them in different ways ("bit-swizzled"). We untangle these once in the Decode stage, and pass on the clarified results in the `imm` field.

[1] If we implement the "C" RISC-V ISA extension (compressed instructions), the correct fall-through PC may be PC+2.

This heterogeneous collection of values is most conveniently expressed as a `struct` type:

src_Common/Inter_Stage.bsv: line 49 ...

```
typedef struct {Bit #(XLEN)  pc;

                Bool         exception;  // Fetch exception/ decode illegal instr
                Bit #(4)     cause;
                Bit #(XLEN)  tval;

                // If not exception
                Bit #(XLEN)  fallthru_pc;
                Bit #(32)    instr;
                OpClass      opclass;
                Bool         has_rs1;
                Bool         has_rs2;
                Bool         has_rd;
                Bool         writes_mem;   // All mem ops other than LOAD
                Bit #(XLEN)  imm;          // Canonical (bit-swizzled)
   ...
} Decode_to_RR
deriving (Bits, FShow);
```

Because we said "`deriving(Bits)`", the *bsc* compiler will automatically work out a representation for `Decode_to_RR` values in bits, using the straightforward method of simply concatenating the bit-vectors of each field into a bit-vector for the whole struct. The total bit-size of a `Decode_to_RR` struct value is simply the sum of the individual bit-sizes of the fields. If we had not said "`deriving(Bits)`", we could explicitly provide some other custom representation in bits.[2]

NOTE:

SystemVerilog makes a distiction beteen "packed" and "unpacked" values.

In BSV all struct and vector values are packed (no padding between fields/elements) unless the user has explicitly over-ridden the "deriving (Bits)" directive with their own bit-representation function.

CAVEAT:

Unfortunately Verilog, the target language for the *bsc* compiler, does not have any concept of structs. When debugging Verilog code that has been produced by *bsc* from BSV source code, a struct will appear as a flat bit-vector that aggregates all the fields.

Because we said "`deriving(FShow)`", the *bsc* compiler will automatically define an "`fshow()`" function for this type: if we print `fshow(`$v$`)`, it will print something like this:

```
Decode_to_RR {pc=..., exception = ..., instr=..., ... }
```

[2] In C/C++, compilers will often "pad" out fields (insert unused bits between fields) to be aligned on byte and word boundaries, for more efficient access in byte-structured memories; thus, a struct's size in C/C++ may be larger than the sum of the field sizes, and may even vary depending on the compiler's target architecture. In hardware design, these values may reside in wires, registers, FIFOs, etc which have no "byte-structured" bias, and so we do not play any such "padding" games.

### 5.2.1 Creating struct values

We can create a new value of type `D_to_RR` with syntax like this:

```
1  Decode_to_RR x = Decode_to_RR {pc:           ... value of field ... ,
2                                 exception:    ... value of field ... ,
3                                 cause:        ... value of field ... ,
4                                 fallthru_pc:  ... value of field ... ,
5                                 instr:        ... value of field ... ,
6                                 ...};
```

The right-hand side is sometimes called a "struct expression", *i.e.,* it is an expression which, when evaluated, produces a struct value.

The repetition of `Decode_to_RR` above seems verbose; the left-hand side instance is the type, and the right-hand side instance is the "struct constructor" (think of it as a function that takes the field values as arguments and returns a struct value). The *bsc* compiler's type-analysis is able to infer the type from the right-hand side, so we can just use the keyword "`let`":

```
1  let x = Decode_to_RR {pc:           ... value of field ... ,
2                        exception:    ... value of field ... ,
3                        cause:        ... value of field ... ,
4                        fallthru_pc:  ... value of field ... ,
5                        instr:        ... value of field ... ,
6                        ...};
```

The order in which the field values are given does not matter; the *bsc* compiler will put the fields into the correct offsets in the struct value.

### 5.2.2 BSV: Don't-care values

Not all field values need be given in a struct expression. The *bsc* compiler will issue a warning for each unspecified field, and insert an "unspecified" (and unpredictable) value there. You can indicate that a field is intentionally left unspecified (and suppress the compiler warning) using "`?`", BSV's notation for a "don't care" value:

```
1     let x = Decode_to_RR {pc:           ... ,
2                           exception:    False,
3                           cause:        ?,
4                           fallthru_pc:  ... ,
5                           instr:        ... ,
6                           ...};
```

In the above example, the exception `cause` field is meaningless when the `exception` field is false, and we indicate this explicitly with "`?`".

"Don't care" values are useful for several reasons. First, this conveys to the human reader that the value in this field is irrelevant.

Second, it can result in more efficient circuitry. If we had said "0", for example, the *bsc* compiler has to create circuitry ensuring that field's value is 0. By saying "`?`", the *bsc* compiler is allowed to omit all that circuitry.

Third, in places where it does not result in additional hardware, the *bsc* compiler usually injects the specific value `'h_AAAA_AAAA` (of suitable bit-width). While debugging, observing such a value in some computation is often a clue that something is wrong (it roughly plays the role of "X" values in Verilog/SystemVerilog/VHDL).

NOTE:

> Verilog, SystemVerilog and VHDL have a concept of "X" values. Each bit of a register or wire carries an "X" value until it has been assigned a specific binary value (0 or 1). However, note that this is *only in simulation*, where the simulator can and does model 3-valued logic (0, 1 and X) for each bit, and is able to propagate X values through operators, registers, *etc.* Hardware only implements 2-valued logic—every bit is either 0 or 1. Thus, this is an artefact that is only useful during debugging in simulation and static analysis.
>
> BSV only has 2-valued logic; there is no concept of an "X" value. A BSV "`?`" expression has some specific, but potentially unpredictable, binary value.

### 5.2.3 Selecting struct fields

Struct fields can be selected using the usual "dot" notation common to SystemVerilog and C/C++:

```
1   x.pc
2   x.instr
```

### 5.2.4 Updating struct fields using assignment

Struct fields can be updated with assignment using the usual "dot" notation common to SystemVerilog and C/C++:

```
1   x.pc    = ... new value ... ;
2   x.instr = ... new value ... ;
```

## 5.3 RISC-V: Memory Requests and Responses; IMem and DMem

In Figure 5.1, the `Mem_Req` and `Mem_Rsp` structs are used in two places. The Fetch step issues a memory request, and the corresponding memory response is received by the Decode step. Similarly, the "Execute Memory Ops" steps issues a memory request and consumes a memory response. We use the shorthand term "IMem" for the first context (for Instruction Memory) and "DMem" for the latter context (for Data Memory).

### 5.3.1 Separation of IMem and DMem (Harvard Architecture)

The separation of memory channels for instructions and data (loads/stores) is quite standard in modern CPU architectures, and is informally called a "Harvard Architecture". The term refers to the architecture of the Harvard Mark I computer, designed and built by Harvard University and IBM in the 1940s (the term itself was coined much later). It sometimes refers just to separate, concurrent paths to memory for instructions and data, and sometimes also to physically separate memories for instructions and data (more discussion in Wikipedia: https://en.wikipedia.org/wiki/Harvard_architecture).

Modern software is typically not "self-modifying", *i.e.,* instructions and data are placed in different areas of memory, and load/store instructions never write into the instruction area, *i.e.,* programs never over-write instructions in memory. This allows separate hardware for memory access for instructions *vs.* memory access for data, which can run concurrently, *i.e.,* we may fetch an instruction at the same time as we are accessing data memory for a previous load/store instruction (we will see this in Fife). We can also tune and optimize each memory path separately for their different dynamic behavioral patterns. In some systems we can also *protect* the instruction memory area, *i.e.,* enforce in hardware the policy of not over-writing instructions.

This view of strict separation of IMem and DMem has to be tempered somewhat when considering languages like JavaScript, Python *etc.* that employ so-called "JIT" compiling ("Just-In-Time"). The run-time systems of such languages generate instructions on-the-fly, *i.e.,*, LOAD/STORE instructions produce *data* through the DMem channel that will (soon) be fetched as *instructions*. But even in these systems, there is a strict protocol of *phases*. During a code-generation phase, the data produced is considered as ordinary data. Then there is a deliberately executed phase-change, where the virtual memory protections of the data-pages just written are changed so that they are now viewed as read-only instruction pages, after which these new instructions can be fetched.

### 5.3.2 RISC-V: Memory Requests

A memory-request in RV32I is either a LOAD, a STORE or a FENCE. We could define an enum type for this:

```
1    typedef enum {MEM_REQ_LOAD,
2                  MEM_REQ_STORE
3                  MEM_REQ_FENCE} Mem_Req_Type
4  deriving (Eq, FShow, Bits);
```

which will use a 2-bit encoding (0 for LOAD, 1 for STORE, 2 for FENCE). However, we look to the future where we might extend Drum and Fife to implement the "A" extension (for "Atomic Memory Operations"). The RISC-V ISA Unprivileged Spec document, Chapter 8, described additional memory operations LR (Load-Reserved), SC (Store-Conditional), AMOSWAP, AMOADD, AMOXOR, AMOAND, AMOOR, AMOMIN, AMOMAX, AMOMINU, and AMOMAXU, each of which comes in a 32-bit version (in RV32 and RV64) and a 64-bit version (in RV64). These operations are coded with 5 bits in the instruction (`instr[31:27]`).

Accordingly, we use a 5-bit encoding for LOAD, STORE and FENCE, using 5-bit codes that are not used in the A extension:

src_Common/Instr_Bits.bsv: line 226 ...

```
Bit #(5) funct5_LOAD    = 5'b_11110;
Bit #(5) funct5_STORE   = 5'b_11111;
Bit #(5) funct5_FENCE   = 5'b_11101;
```

An IMem request is for one 32-bit instruction (four bytes).[3] A DMem request may be for one, two or four bytes. We express these request-size options using an enum type:

src_Common/Mem_Req_Rsp.bsv: line 39 ...

```
typedef enum {MEM_1B, MEM_2B, MEM_4B, MEM_8B} Mem_Req_Size
deriving (Eq, FShow, Bits);
```

A memory request bundles a request type, a size, and an address. For memory-writes, we also bundle the data to be stored. We express this bundle using a struct:

src_Common/Mem_Req_Rsp.bsv: line 42 ...

```
typedef struct {Mem_Req_Type  req_type;
                Mem_Req_Size  size;
                Bit #(64)     addr;
                Bit #(64)     data;     // CPU => mem data
                ...
} Mem_Req
deriving (Eq, FShow, Bits);
```

In `Mem_Req_Size`, the option `Mem_8B` is not possible in RV32I. Similarly, in `Mem_Req` the `addr` and `data` fields need only be 32-bits wide for RV32I. However, we have declared them as shown looking ahead into the future, where we may wish to implement RV64I, or where we may wish to implement the "D" ISA extension (double-precision floating point values, which are represented in 64-bits).

For STORE requests of 1 and 2 bytes (*i.e.,* smaller than the `data` field) we assume the data is passed in the least-significant bytes of the `data` field.

This is the information sent to Memory from the Read-PC-and-Fetch step and also from the Execute-Memory-Ops step in Figure 5.1.

### 5.3.3 RISC-V: Address Alignment

Although nowadays we think of all computer memories in units of 8-bit bytes and being byte-addressed,[4] in practice in hardware, it is usually simpler if memory-requests are *aligned* to an address according to the request size. Specifically, the address for a 2-byte request should be even, *i.e.,* the least significant bit of the address, `addr[0]`, should be zero. The

[3] When implmenting the "C" RISC-V ISA extension (compressed instructions), instructions can also be 16-bits (2 bytes). When implementing more sophisticated Fetch units, we may actually fetch much larger chunks, such as a full cache line.

[4] Some early computers, until about the late 1970s, had other memory granularities—multiples of 6, 7, 9 bits, *etc.* Those were the days of bespoke memories for each computer design. Mass-production of memory chips resulted in standardization to 8-bit bytes.

address for a 4-byte request should have zero in the two least significant bits (`addr[1:0]`) and the address for an 8-byte request should have zero in the three least significant bits (`addr[2:0]`).

We can see why address-alignment is desirable. Memory implementations (chips) are usually architected to retrieve multiple bytes at a time (*e.g.,* 64 bytes) so that all those bytes can share addressing and control circuitry. With such an organization, a misaligned access request may straddle the boundaries of such "naturally sized" units and so may require two consecutive reads/writes. Caches are usually organized to hold multi-byte *cache lines* (*e.g.,*, 32 bytes) in order to share the addressing and miss-handling circuitry, and to move data efficiently in and out of the cache. Again, a misaligned access request may straddle a cache-line boundary, and may require two consecutive accesses, which may hit or miss independently. Virtual memory systems are usually organized in *pages*, units of typically 4K-8K bytes, in order to share virtual-memory handling circuitry, and to move data efficiently between main memory and disks. Again, a misaligned access may straddle a page boundary, and may require two consecutive accesses, which may hit or page-fault independently and differently. In short, misaligned accesses add significant complexity to memory-system hardware design.

We can organize our software so that misaligned accesses are exceedingly rare. Most software is produced by compilers, and the compiler ensures that instructions and data are placed in memory at aligned addresses, possibly by padding gaps between "adjacent" smaller-sized data (such as a pair of 1-byte-sized fields in a struct). This padding may waste a few bytes of memory, but pays back in greater speed and reduced complexity.

Although misaligned accesses are rare, we cannot always guarantee their absence in software, since software can calculate an arbitrary address before performing a memory access. It seems wasteful to have to pay for extra hardware complexity (with attendant loss in overall performance) for such rare cases. In many computer systems, therefore, these rare misaligned accesses are relegated to software handling:

- The memory system simply refuses to handle a misaligned access, and returns a "misaligned" error instead.
- The CPU, receiving such an error response, undergoes a "trap" which directs it to piece of software called an trap-handler (or exception handler). The trap-handler (in software) performs the memory access in multiple smaller pieces (in the worst case, of size 1 byte), each of which is aligned. In other words, the trap-handler "completes" the original memory access before resuming the main-line code that attempted it.

The RISC-V ISA specification does not forbid misaligned accesses nor prescribe how they should be handled. Some implementations will handle it in hardware, and other implementations will return a "misaligned" error and rely on a trap handler to complete the access. Some implementations may only run software where it has been proven to not generate misaligned memory requests, and therefore may not even contain a trap-handler for misaligned accesses.

### 5.3.4 RISC-V: Memory Responses

The response from memory for any request may be to report success, an alignment error, or some other error. Examples of "other errors" are:

- Absence of memory at the given address. For example, although RV32I addresses are 32-bits, which can address 4GiB of memory, we may provision our system with something smaller, say 1 GiB.
- An unsupported operation. *E.g.,* an attempt to write into a read-only memory (ROM).
- Corruption of data, due to electrical glitches, environmental electromagnetic pulses, *etc.*. These errors are usually detected with some kind of error-detecting code, such as parity bits.

These different memory response-types can be encoded in an enum type:

src_Common/Mem_Req_Rsp.bsv: line 56 ...

```
typedef enum {MEM_RSP_OK,
              MEM_RSP_MISALIGNED,
              MEM_RSP_ERR,
              ...
} Mem_Rsp_Type
deriving (Eq, FShow, Bits);
```

A memory-response contains the response-type. For a LOAD request with an OK response, it also contains the data that was read from memory. This can be expressed in a struct:

src_Common/Mem_Req_Rsp.bsv: line 65 ...

```
typedef struct {Mem_Rsp_Type  rsp_type;
                Bit #(64)     data;      // mem => CPU data
                ...
} Mem_Rsp
deriving (Eq, FShow, Bits);
```

For LOAD requests of 1 and 2 bytes (*i.e.,* smaller than the `data` field) we assume the data is returned in the least-significant bytes of the `data` field.

# Chapter 6

# RISC-V: Core functions for RISC-V ISA execution (used in Drum and Fife)

## 6.1 Introduction

In this chapter we discuss the core functions of Figure 6.1: `fn_Fetch`, `fn_Decode`, `fn_Dispatch`, `fn_EX_Control` and `fn_EX_Int`.

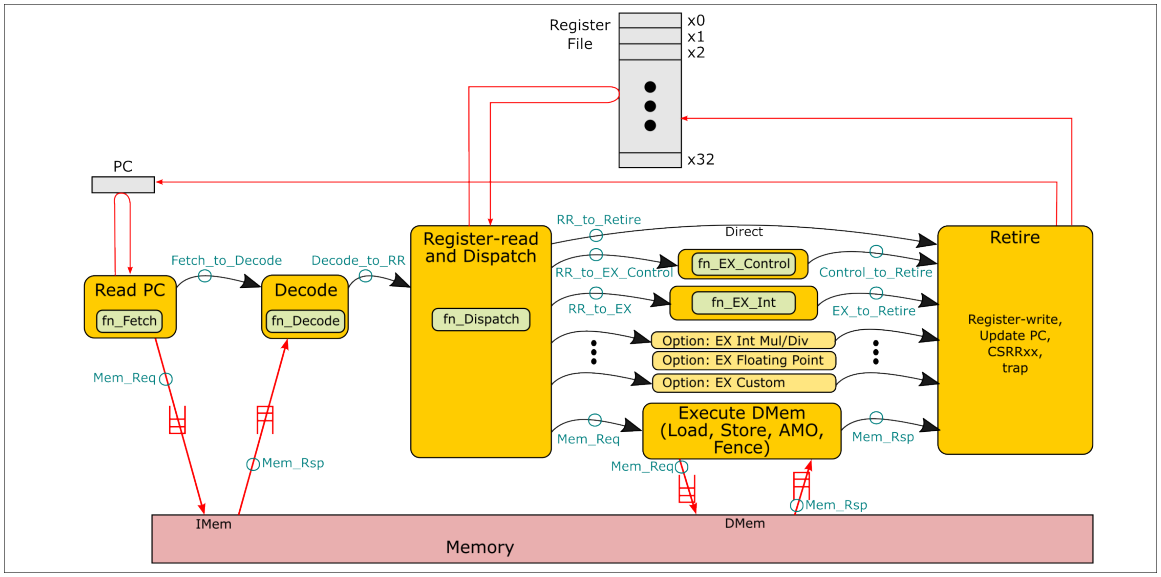



Figure 6.1: Simple interpretation of RISC-V instructions

## 6.2 The function `fn_Fetch`

The Fetch function *per se* is fairly simple, even trivial. Its input is the current value of the program counter (PC), which is used as the address in a memory-request to IMem. It has two outputs,

- A *memory request* to memory, to read an instruction. We have already seen the definition of the `Mem_Req` struct in Section 5.3.2.
- Some additional information "`Fetch_to_Decode`" passed on to the Decode step.

The "`Fetch_to_Decode`" struct has only one interesting field, the PC:

src_Common/Inter_Stage.bsv: line 27 ...

```
typedef struct {
   Bit #(XLEN)  pc;
   ...
   Bit #(64)    inum;            // for debugging only
} Fetch_to_Decode
deriving (Bits, FShow);
```

The field `inum` holds the "instruction number", which is a sequence number counting every instruction fetched. It is used only for debugging, to be able to identify a specific fetched instruction. (In the source code you will see two more fields `predicted_pc` and `epoch`; ignore these for now, they are not used in Drum, only in Fife.)

To pass both results of `fn_Fetch`, we simply use a *nested* struct, *i.e.,* a struct containing the two component structs:

src_Common/Fn_Fetch.bsv: line 25 ...

```
typedef struct {
   Fetch_to_Decode   to_D;
   Mem_Req  mem_req;
} Result_F
deriving (Bits, FShow);
```

Finally, as mentioned earlier, the function `fn_Fetch` is almost trivial: it merely fills in the two result structs based on argument values and returns them:

src_Common/Fn_Fetch.bsv: line 31 ...

```
// This is actually a pure function; is ActionValue only to allow
// $display insertion for debugging
function ActionValue #(Result_F)
         fn_Fetch (Bit #(XLEN)  pc,
                   ...
                   Bit #(64)    inum,
                   ...
   actionvalue
      Result_F y = ?;
      // Info to next stage
      y.to_D = Fetch_to_Decode {pc:           pc,
                                ...
      // Request to IMem
      y.mem_req = Mem_Req {req_type: funct5_LOAD,
                           size:     MEM_4B,
                           addr:     zeroExtend (pc),
                           data :    ?,
                           // Debugging
                           inum:     inum,
                           ...
      return y;
```

```
22     endactionvalue
23 endfunction
```

As the comment at the top of the code excerpt says, `fn_Fetch` is actually a pure function (no side-effects), and we could have written it as follows:

```
1  function Result_F
2          fn_Fetch (Bit #(XLEN)  pc,
3                    Bit #(XLEN)  predicted_pc,    // \belide{19}
4                    Epoch        epoch,
5                    File         flog,
6                    Bit #(64)    inum);           // \eelide
7        Result_F y = ?;
8        ...
9        return y;
10 endfunction
```

*i.e.,* the return-type is `Result_F` instead of `ActionValue#(Result_F)`, and we simply drop the "`actionvalue`—`endactionvalue`" bracket keywords. Recall Section 4.6.1 where we discussed pure *vs.* side-effecting functions, and their distinction in the type-system through the absence/presence of the `ActionValue` type constructor. Even though this is a pure function, by couching it as an `ActionValue` type, we preserve the option of inserting a `$display` (which *is* a side-effect) for debugging purposes, should we need it in the future.

In line 9 we declare variable `y` of type `Result_F`, with unspecified initial value (recall Section 5.2.2 where we discussed using "`?`" to indicate an unspecified or don't-care value). The two fields of `y` are then updated in lines 11 and 14, respectively, and `y` is finally returned as the result of the function.

We also use "`?`" in line 17 the `data` field is only relevant in a STORE memory request, whereas this is a FETCH request.

---

**Exercise 6.1:**

Write a testbench for `fn_Fetch()`, apply it to a number of 32-bit values (PC values) and print the results using `$display` and `fshow`, and visually check that the `Fetch_to_Decode` and `Mem_Req` outputs look correct.

□

---

## 6.3 The `fn_Decode` function

The core function for the Decode step is called `fn_Decode`. Its arguments are a struct of type `Fetch_to_Decode` from the Fetch step and a `Mem_Rsp` memory response struct from memory. Its output struct type, `Decode_to_RR`, was described in Section 5.2. The code for `fn_Decode` is mostly a big if-then-else that analyses the incoming instruction and produces some summary information:

src_Common/Fn_Decode.bsv: line 28 ...

```
function ActionValue #(Decode_to_RR)
        fn_Decode (Fetch_to_Decode  x_F_to_D,
                   Mem_Rsp          rsp_IMem,
                   ...
   actionvalue
      Bit #(32) instr = truncate (rsp_IMem.data);
      Bit #(5)  rd    = instr_rd (instr);

      let fallthru_pc = x_F_to_D.pc + 4;

      // Baseline info to next stage
      let y = Decode_to_RR {pc:           x_F_to_D.pc,

                            exception:    False,
                            cause:        ?,
                            tval:         0,

                            // not-exception
                            fallthru_pc:  fallthru_pc,
                            instr:        instr,
                            opclass:      ?,
                            has_rs1:      False,
                            has_rs2:      False,
                            has_rd:       False,
                            writes_mem:   False,
                            imm:          0,
                            ...

      Bool non_zero_rd = (rd != 0);

      if (rsp_IMem.rsp_type == MEM_RSP_MISALIGNED) begin
         y.exception = True;
         y.cause     = cause_INSTRUCTION_ADDRESS_MISALIGNED;
         y.tval      = truncate (rsp_IMem.addr);
      end
      else if (rsp_IMem.rsp_type == MEM_RSP_ERR) begin
         y.exception = True;
         y.cause     = cause_INSTRUCTION_ACCESS_FAULT;
         y.tval      = truncate (rsp_IMem.addr);
      end
      ...
      else if (is_legal_LUI (instr) || is_legal_AUIPC (instr)) begin
         y.opclass = OPCLASS_INT;
         y.has_rd  = non_zero_rd;
         y.imm     = signExtend ({ instr_imm_U (instr), 12'h000 });
      end
      else if (is_legal_BRANCH (instr)) begin
         y.opclass = OPCLASS_CONTROL;
         y.has_rs1 = True;
         y.has_rs2 = True;
         y.imm     = signExtend (instr_imm_B (instr));
      end
      else if (is_legal_JAL (instr)) begin
```

```
54          y.opclass = OPCLASS_CONTROL;
55          y.has_rd  = non_zero_rd;
56          y.imm     = signExtend (instr_imm_J (instr));
57       end
58       else if (is_legal_JALR (instr)) begin
59          y.opclass = OPCLASS_CONTROL;
60          y.has_rs1 = True;
61          y.has_rd  = non_zero_rd;
62          y.imm     = signExtend (instr_imm_I (instr));
63       end
64       else if (is_legal_LOAD (instr)) begin
65          y.opclass = OPCLASS_MEM;
66          y.has_rs1 = True;
67          y.has_rd  = non_zero_rd;
68          y.imm     = signExtend (instr_imm_I (instr));
69       end
70       else if (is_legal_STORE (instr)) begin
71          y.opclass    = OPCLASS_MEM;
72          y.has_rs1    = True;
73          y.has_rs2    = True;
74          y.writes_mem = True;
75          y.imm        = signExtend (instr_imm_S (instr));
76       end
77       else if (is_legal_OP_IMM (instr)) begin
78          y.opclass = OPCLASS_INT;
79          y.has_rs1 = True;
80          y.has_rd  = non_zero_rd;
81          y.imm     = signExtend (instr_imm_I (instr));
82       end
83       else if (is_legal_OP (instr)) begin
84          y.opclass = OPCLASS_INT;
85          y.has_rs1 = True;
86          y.has_rs2 = True;
87          y.has_rd  = non_zero_rd;
88       end
89       else if (is_legal_ECALL (instr)
90                || is_legal_EBREAK (instr)
91                || is_legal_MRET (instr)) begin
92          y.opclass = OPCLASS_SYSTEM;
93       end
94       else if (is_legal_CSRRxx (instr)) begin
95          y.opclass = OPCLASS_SYSTEM;
96          y.has_rs1 = (instr_funct3 (instr) [2] == 0);
97          y.has_rd  = non_zero_rd;
98       end
99       else if (is_legal_FENCE (instr)) begin
100         y.opclass = OPCLASS_FENCE;
101      end
102      else begin
103         y.exception = True;
104         y.cause     = cause_ILLEGAL_INSTRUCTION;
105         y.tval      = truncate (instr);
106      end
107
```

```
108         return y;
109      endactionvalue
110   endfunction
```

In line 6, we extract the instruction from the `Mem_Rsp` memory response `data` from the Fetch operation. The `truncate` operation is used to shrink the bit-vector width of `rsp_IMem.data` (64 bits) to the bit-vector width of `instr` (32 bits). (Section 5.3.2 discussed why we declared `rsp_IMem.data` to be 64-bits wide). The `truncate` operation is polymorphic, accepting arguments of any bit-width that is at least as wide as the required output. Note, `truncate` keeps least-significant bits and drops most-significant bits.

In line 7 we extract the `rd` ("destination register") field from the instruction. In line 9 we compute the fall-through PC, PC+4 (with the caveat that if extend the implementation to support the "C" RISC-V ISA extension ("Compressed" instructions), it may be PC+2, which information can be gleaned from the instruction encoding).

In lines 11-26 we create a baseline `Decode_to_RR` value which we will selectively modify in the if-then-else statements that follow.

In lines 31-40 we first handle the sitations where the Fetch operation to memory itself returned an error. We mark the `exception` field True and fill in the appropriate `cause` and `tval` (these will be placed in MCAUSE and MTVAL CSRs in the Retire step).

The rest of the code is a series of if-then-else clauses. Each clause identifies one class of instruction and updates the `opclass` field correspondingly. The repertoire of instructions that we consider are the forty instructions listed in the "RV32I Base Instruction Set" table of "Table 24.2: Instruction listing for RISC-V" of the Unprivileged Spec [15].

Each if-then-else clause also fills in the `has_rs1`, `has_rs2` and `has_rd` fields, as appropriate, for each class of instruction. Note that the `has_rd` field is set to False if `rd` is zero (recall that general-purpose register `x0` ignores writes and always reads as 0).

We also decode each kind of "immediate" and fill in the `imm` field in the struct. Recall from Section 2.5, including Figures 2.4 and 2.5, that different classes of instructions encode immediate values in different ways, and the immediate values can have different bit-widths. We use the functions `instr_imm_I()`, `instr_imm_S()`, `instr_imm_B()`, `instr_imm_U()` and `instr_imm_J()` to extract the and rearrange the immediate bits appropriately for each class of instruction. Then, here in `fn_Decode`, we zero- or sign-extend each immediate as appropriate so that, from this point onwards, each immediate can be treated as an ordinary `Bit#(XLEN)` value.

An exercise below suggests that you write the code for these `instr_imm_X` functions; it's good practice for the BSV beginner!

The final "else" clause is selected if the instruction does not match any of the forty RV32I instructions. In this case we set the `exception` field, and set the `cause` field to indicate an illegal instruction.

Observe that the entire `fn_Decode()` function is just a (large) combinational circuit—it is an acyclic composition of smaller combinational circuits, many of which we've seen earlier. The whole `fn_Decode()` function can be visualized as a box with incoming wires corresponding to the `Fetch_to_Decode` struct and the `Mem_Rsp` struct, outgoing wires corresponding to

the `Decode_to_RR` struct, and filled with logic gates that compute each output wire as a function of the input wires.

---

**Exercise 6.2:**

The provided source code includes the functions `instr_imm_I()`, `instr_imm_S()`, `instr_imm_B()`, `instr_imm_U()` and `instr_imm_J()` (in file `Instr_Bits.bsv`), but try to write them yourself first, and compare your solutions to the provided codes.

**Exercise 6.3:**

Write a testbench for `fn_Decode()`, apply it to a number of PC and instruction values. For each input PC value, construct an `Fetch_to_Decode` struct around it. For each input instruction, construct a `Mem_Rsp` struct around it, some with memory errors, some without. Apply `fn_Decode` to such pairs. Print the results results using `$display` and `fshow`, and visually check that the `D_to_RR` outputs look correct.

□

---

## 6.4 The `fn_Dispatch` function after reading input registers

In the Register-Read and Dispatch step ("RR", "RRD"), we first read the `rs1` and `rs2` values from the Register File. We will cover register files in Section 7.4, and register-reads when we discuss the Drum CPU itself, in Chapter 10.

With the `rs1` and `rs2` values in hand, we use function `fn_Dispatch` to determine the information to be passed to the various alternative steps ("flows") that follow. Its argument is a `Decode_to_RR` struct from the Decode step, which was discussed in Section 5.2. Before we look at the result type of `fn_Dispatch`, let us discuss the four possible subsequent flows:

- "Direct": Some information is sent directly from RR to Retire for *every* instruction. Crucially, RR sends a *tag* that indicates which of the four flows is being followed for the current instruction.

  Additional direct information includes information from RR (PC, whether an exception has already been seen in the Fetch or Decode stages, `has_rd`, `writes_mem`, the instruction, the fall-through PC and the `rs1` value (needed for CSRRx instructions).

  The direct flow is also used for all SYSTEM instructions, including CSRRxx, ECALL, EBREAK, and MRET.

- "Control": If the instruction is a BRANCH, JAL or JALR, we produce information for the Execute Control flow.

- "Integer": If the instruction is a LUI, AUIPC or integer arithmetic or logic (IALU) operation, we produce information for the Execute Integer flow.

- "DMem": If the instruction is a LOAD, STORE or FENCE, we produce information for the Execute Memory flow.

The following enum type declaration defines four constants to identify the flow for this instruction:

src_Common/Inter_Stage.bsv: line 75 ...

```
typedef enum {EXEC_TAG_DIRECT,
              EXEC_TAG_CONTROL,
              EXEC_TAG_INT,
              EXEC_TAG_DMEM
} Exec_Tag
deriving (Bits, Eq, FShow);
```

The result type of `fn_Dispatch` is `Result_Dispatch`. It is just a nested struct that contains four different struct types for the four flows (the struct types can be seen on arc labels in Figure 6.1).

src_Common/Fn_Dispatch.bsv: line 29 ...

```
typedef struct {
   RR_to_Retire      to_Retire;
   RR_to_EX_Control  to_EX_Control;
   RR_to_EX          to_EX;
   Mem_Req           to_EX_DMem;
} Result_Dispatch
deriving (Bits, FShow);
```

The first component is the information sent directly to Retire:

src_Common/Inter_Stage.bsv: line 82 ...

```
typedef struct {Exec_Tag     exec_tag;    // ``flow'' for this instr

                Bit #(XLEN)  pc;
                Bool         has_rd;      // From RR
                Bool         writes_mem;  // From RR

                Bool         exception;   // Fetch exception, decode illegal instr
                Bit #(4)     cause;
                Bit #(XLEN)  tval;

                // If not exception
                Bit #(32)    instr;
                Bit #(XLEN)  fallthru_pc;
                Bit #(XLEN)  rs1_val;     // For CSRRXX instrs
                ...


                Bit #(64)    inum;            // for debugging only
} RR_to_Retire
deriving (Bits, FShow);
```

The `exec_tag` informs Retire about the flow for this instruction. The `exception` and `cause` fields from `Decode_to_RR` are carried through, as-is, since it is the Retire step that handles all exceptions. Note, in addition to passing these exceptions in RR to Retire, the Control or Execute steps can also produce exceptions. The `pc` field is needed in case Retire needs to handle a trap or interrupt, which saves the `pc` before handling it.

The `has_rd` field is carried through to control whether Retire tries to write a value back to the register file nor not. The `fallthru_pc` is used for most instructions that complete successfully (without raising an exception).

The second component of `Result_Dispatch` is the information sent to Execute Control (we will see how these fields are used in Section 6.5 when we discuss `fn_EX_Control`):

src_Common/Inter_Stage.bsv: line 110 ...

```
typedef struct {Bit #(XLEN)  pc;
                Bit #(XLEN)  fallthru_pc;
                Bit #(32)    instr;
                Bit #(XLEN)  rs1_val;
                Bit #(XLEN)  rs2_val;
                Bit #(XLEN)  imm;
                Bit #(64)    inum;    // for debugging only
} RR_to_EX_Control
deriving (Bits, FShow);
```

The third component of `Result_Dispatch` is the information sent to Execute Int (we will see how these fields are used in Section 6.6 when we discuss `fn_EX_Int`):

src_Common/Inter_Stage.bsv: line 141 ...

```
typedef struct {Bit #(32)    instr;
                Bit #(XLEN)  rs1_val;
                Bit #(XLEN)  rs2_val;
                Bit #(XLEN)  imm;
                ...
} RR_to_EX
deriving (Bits, FShow);
```

We choose to call it `RR_to_EX` instead of `RR_to_EX_Int` because the same struct type is likely to be used in future also for the other optional execution pipes shown in Figure 6.1: Integer Multiply Divide ("M" ISA extension), floating point ("F" and "D" ISA extensions) and Custom (non-standard ISA extensions).

The third component of `Result_Dispatch` is the information sent to Execute DMem, and is the same `Mem_Req` struct we have already discussed in Section 5.3.2, and which is also an output component of `fn_Fetch`.

The code for `fn_Dispatch` is shown below. Its arguments are the `Decode_to_RR` struct from the Decode stage, and values from the source registers `rs1` and `rs2`.

src_Common/Fn_Dispatch.bsv: line 39 ...

```
function ActionValue #(Result_Dispatch)
         fn_Dispatch (Decode_to_RR         x,
                      Bit #(XLEN)          rs1_val,
                      Bit #(XLEN)          rs2_val,
                      ...
   actionvalue
      // Compute tag to control merging at Retire
      Exec_Tag exec_tag = EXEC_TAG_DIRECT;    // exceptions and OPCLASS_SYSTEM
      if (! x.exception) begin
         if      (x.opclass == OPCLASS_CONTROL) exec_tag = EXEC_TAG_CONTROL;
         else if (x.opclass == OPCLASS_INT)     exec_tag = EXEC_TAG_INT;
```

```
12          else if (x.opclass == OPCLASS_MEM)     exec_tag = EXEC_TAG_DMEM;
13          else if (x.opclass == OPCLASS_FENCE)   exec_tag = EXEC_TAG_DMEM;
14       end
15
16       let to_Retire = RR_to_Retire {exec_tag:     exec_tag,
17
18                                     pc:           x.pc,
19                                     has_rd:       x.has_rd,
20                                     writes_mem:   x.writes_mem,
21
22                                     exception:    x.exception,
23                                     cause:        x.cause,
24                                     tval:         x.tval,
25
26                                     instr:        x.instr,
27                                     fallthru_pc:  x.fallthru_pc,
28                                     rs1_val:      rs1_val,
29                                     ...
30
31       // ----------------
32       // Info for EX_Control
33       let to_EX_Control = RR_to_EX_Control {pc:           x.pc,
34                                             fallthru_pc:  x.fallthru_pc,
35                                             instr:        x.instr,
36                                             rs1_val:      rs1_val,
37                                             rs2_val:      rs2_val,
38                                             imm:          x.imm,
39                                             ...
40       // ----------------
41       // Info for Execute Int pipe
42       let to_EX  = RR_to_EX {pc:      x.pc,
43                              instr:   x.instr,
44                              rs1_val: rs1_val,
45                              rs2_val: rs2_val,
46                              imm:     x.imm,
47                              ...
48       // ----------------
49       // Info for Execute DMem pipe
50       Bit #(XLEN)  eaddr    = rs1_val + x.imm;
51       Mem_Req_Size mrq_size = unpack (x.instr [13:12]);  // B, H, W or D
52       Mem_Req_Type mrq_type = (is_LOAD (x.instr) ? funct5_LOAD
53                                : (is_STORE (x.instr) ? funct5_STORE
54                                   : (is_FENCE (x.instr) ? funct5_FENCE
55                                      : funct5_BOGUS)));
56
57       let to_EX_DMem = Mem_Req {req_type: mrq_type,
58                                 size:     mrq_size,
59                                 addr:     zeroExtend (eaddr),
60                                 data:     zeroExtend (rs2_val),
61                                 ...
62
63       // ----------------
64       // Construct and return final result
65       let result = Result_Dispatch {to_Retire:     to_Retire,
```

```
66                                      to_EX_Control: to_EX_Control,
67                                      to_EX:         to_EX,
68                                      to_EX_DMem:    to_EX_DMem};
69      return result;
70   endactionvalue
71 endfunction
```

Lines 7-14 compute `exec_tag`, *i.e.,* the flow to be followed.

Lines 16-29, 33-38, 42-46 and 49-61 construct the flow-specific struct values. The latter three are meaningful only if `exec_tag` indicates Control, Integer of DMem, respectively, but there is no harm in constructing them, even if they may contain bogus data; they will only be used when their flows are chosen and ignored otherwise.

Lines 64-68 construct and return the final result with the four component structs.

## 6.5 The `fn_EX_Control` function

This function is a simple one-input one-output function. The input type `RR_to_EX_Control` was described in Section 6.4, where it was an output type. The output type of `fn_EX_Control` is shown below:

src_Common/Inter_Stage.bsv: line 121 ...

```
1 typedef struct {Bool         exception;  // Misaligned BRANCH/JAL/JALR target
2                 Bit #(4)     cause;
3                 Bit #(XLEN)  tval;
4
5                 Bit #(XLEN)  next_pc;
6                 Bit #(XLEN)  data;          // Return-PC for JAL/JALR
7                 ...
8 } EX_Control_to_Retire
9 deriving (Bits, FShow);
```

Here is the Execute Control function:

src_Common/Fn_EX_Control.bsv: line 32 ...

```
1  function ActionValue #(EX_Control_to_Retire)
2           fn_EX_Control (RR_to_EX_Control  x,
3                          ...
4       Bit #(XLEN)  next_pc   = ?;
5       Bool         exception = False;    // Misaligned target_pc
6
7       if (is_BRANCH (instr)) begin
8          Bool branch_taken = case (instr_funct3 (instr))
9                                 funct3_BEQ:  (rs1_val == rs2_val);
10                                funct3_BNE:  (rs1_val != rs2_val);
11                                funct3_BLT:  signedLT (rs1_val, rs2_val);
12                                funct3_BGE:  signedGE (rs1_val, rs2_val);
13                                funct3_BLTU: (rs1_val < rs2_val);
14                                funct3_BGEU: (rs1_val >= rs2_val);
15                              endcase;
16         let target_pc = x.pc + x.imm;
```

```
17          next_pc = (branch_taken ? target_pc : x.fallthru_pc);
18          exception = (branch_taken && (target_pc [1:0] != 0));
19          ...
20       end
21       else if (is_JAL (instr)) begin
22          next_pc = x.pc + x.imm;
23          exception = (next_pc [1:0] != 0);
24          ...
25       end
26       else if (is_JALR (instr)) begin
27          // zero out LSB in target PC
28          next_pc = ((rs1_val + x.imm) & ~1);
29          exception = (next_pc [1:0] != 0);
30          ...
31       end
32       ...
33
34       let y = EX_Control_to_Retire {exception:  exception,
35                                     cause:      cause_INSTRUCTION_ADDRESS_MISALIGNED,
36                                     tval:       x.pc,
37                                     next_pc:    next_pc,
38                                     data:       x.fallthru_pc,
39                                     ...
40       return y;
41    endactionvalue
42 endfunction
```

The first "`if`" clause handles BRANCH (conditional branch) instructions. The `case` expression computes the boolean value `branch_taken`, *i.e.,* the decision whether to take the branch or not. A case-clause is chosen based on the 3-bit `funct3` field of the instruction that identifies the specific condition to be tested.[1] Note that for BLT and BGE, we use BSV library functions `signedLT` and `signedGE` that interpret `rs1_val` and `rs2_val` as 2's-complement signed integers.

Line 16 computes the target PC should the the branch be taken. Line 17 computes the next PC, which is either the target PC or the fall-through PC depending on whether or not the branch is taken. Finally, Line 18 checks, if the branch is taken, that the target PC is a properly aligned address (if not, we must raise an exception).

The next "`else if`" clause handles JAL (Jump and Link) instructions and the final "`else if`" clause handles the JALR (Jump and Link Register) instructions. They are both straightforward, unconditional calculations of a next PC, along with an alignment-check that the next PC is suitably aligned. As per the Unprivileged ISA specification document, JALR also zeroes the least-significant bit of target PC.

The final section constructs the `EX_Control_to_Retire` struct result and returns it.

---

**Exercise 6.4:**

[1] Unlike C/C++, where a case-clause "falls through" to the next case-clause unless you have a `break` statement, in BSV only one case-clause is executed, there is no fall-through.

In `fn_EX_Control`, the BRANCH, JAL and JALR clauses set the `exception` field to true if the next PC is not aligned.

1. What would happen if we did not set the exception field here?
2. See "Section 2.5 Control Transfer Instructions" in the Unprivileged ISA specification document [15]) for a discussion of why we set it here.

**Exercise 6.5:**

Prove (informally) that the three-way if-then-else shown above in `Fn_EX_Control` will catch all cases, *i.e.,* that we never need a final "`else`" clause. This requires reviewing `fn_Decode` and tracking the flow of information through the Register-Read-and-Dispatch step (including `fn_Dispatch`), into `fn_Control`.

**Exercise 6.6:**

In `Fn_EX_Control`, can we change the final "`else if`":

```
   else if (is_JALR (instr)) begin
```

into a simple "`else`", *i.e.,* omit the the `is_JALR` check?

```
   else begin
```

What might be the hardware implication of such a change?

**Exercise 6.7:**

In the final section of `fn_EX_Control`, why do we have: `data: x.fallthrough_pc`?
*Hint*: review the semantics of JAL and JALR instructions.

What if this is a BRANCH instruction?

What if this is a JAL or JALR instruction with `rd` = 0?

□

---

## 6.6 The `fn_EX_Int` function

This function is a simple one-input one-output function. The input type `RR_to_EX` was described in Section 6.4, where it was an output type. The output type of `fn_EX_Int` is shown below:

src_Common/Inter_Stage.bsv: line 154 ...

```
1 typedef struct {Bool         exception;
2                 Bit #(4)     cause;
3                 Bit #(XLEN)  tval;
4
5                 Bit #(XLEN)  data;
6                 ...
7 } EX_to_Retire
8 deriving (Bits, FShow);
```

We choose to call it `EX_to_Retire` instead of `EX_Int_to_Retire` because the same struct type is likely to be used in future also for the other optional execution pipes shown in Figure 6.1: Integer Multiply Divide ("M" ISA extension), floating point ("F" and "D" ISA extensions) and Custom (non-standard ISA extensions).

Here is the Execute Integer function:

src_Common/Fn_EX_Int.bsv: line 31 ...

```
function ActionValue #(EX_to_Retire)
         fn_EX_Int (RR_to_EX  x,
                    ...
   actionvalue
      let instr = x.instr;

      let y = EX_to_Retire {exception:  False,
                            cause:      ?,
                            tval:       ?,
                            data:       ?,
                            ...

      if (is_LUI (instr)) begin
         y.data = x.imm;
         ...
      end
      else if (is_AUIPC (instr)) begin
         y.data = x.pc + x.imm;
         ...
      end
      else begin
         let result <- fn_IALU (instr, x.rs1_val, x.rs2_val, x.imm,
                                ...
         y.data = result;
         ...
      end
      return y;
   endactionvalue
endfunction
```

The first "`if`" handles LUI instructions (Load Upper Immediate). The following "`else if`" handles AUIPC instructions (Add Upper Immediate to PC). The final "`else`" handles all the remaining Integer ops by invoking an "ALU" function:

src_Common/IALU.bsv: line 28 ...

```
function ActionValue #(Bit #(XLEN))
         fn_IALU (Bit #(32)    instr,
                  Bit #(XLEN)  v1,
                  Bit #(XLEN)  v2,
                  Bit #(32)    imm,
                  ...
   actionvalue
      Bit #(7)    opcode = instr_opcode (instr);
      Bit #(3)    funct3 = instr_funct3 (instr);
      // Signed int versions of v1, v2 and imm
      Int #(XLEN) iv1    = unpack (v1);
```

```
12       Int #(XLEN) iv2    = unpack (v2);
13       Int #(XLEN) i_imm  = unpack (imm);
14       ...
15       Bit #(XLEN) y_OP     = 0;
16       if (opcode == opcode_OP) begin
17          ...
18          Bit #(6) shamt = (v2 [5:0] & ((xlen == 32) ? 'h1F : 'h3F));
19          case (funct3)
20             funct3_ADD:  y_OP = pack ((instr [30] == 1'b0)
21                                       ? (iv1 + iv2)
22                                       : (iv1 - iv2));
23             funct3_SLL:  y_OP = v1 << shamt;
24             funct3_SLT:  y_OP = ((iv1 < iv2) ? 1 : 0);
25             funct3_SLTU: y_OP = ((v1  < v2)  ? 1 : 0);
26             funct3_XOR:  y_OP = v1 ^ v2;
27             funct3_SRL:  y_OP = v1 >> shamt;
28             funct3_SRA:  y_OP = pack (iv1 >> shamt);
29             funct3_OR:   y_OP = v1 | v2;
30             funct3_AND:  y_OP = v1 & v2;
31             ...
32          endcase
33          ...
34       end
35
36       Bit #(XLEN) y_OP_IMM = 0;
37       if (opcode == opcode_OP_IMM) begin
38          ...
39          Bit #(6) shamt = (imm [5:0] & ((xlen == 32) ? 'h1F : 'h3F));
40          case (funct3)
41             funct3_ADDI:  y_OP_IMM = pack (iv1 + i_imm);
42             funct3_SLTI:  y_OP_IMM = ((iv1 < i_imm) ? 1 : 0);
43             funct3_SLTIU: y_OP_IMM = ((v1  < imm)   ? 1 : 0);
44             funct3_XORI:  y_OP_IMM = v1 ^ imm;
45             funct3_ORI:   y_OP_IMM = v1 | imm;
46             funct3_ANDI:  y_OP_IMM = v1 & imm;
47             funct3_SLLI:  y_OP_IMM = v1 << shamt;
48             funct3_SRLI:  y_OP_IMM = v1 >> shamt;
49             funct3_SRAI:  y_OP_IMM = pack (iv1 >> shamt);
50             ...
51          endcase
52          ...
53       end
54
55       Bit #(XLEN) result = y_OP | y_OP_IMM;
56       ...
57       return result;
58    endactionvalue
59 endfunction
```

The reason we create a separate `fn_IALU` function, instead of inlining it into `Fn_EX_Int` is because `fn_IALU` is not very RISC-V specific, and may be useful in other contexts that have nothing to do with RISC-V, Drum or FIfe.

In lines 11-13, we define signed-integer versions `iv1`, `iv2` and `i_imm` of the unsigned integer

values `v1`, `v2`, and `imm`, respectively, using the standard BSV `unpack` function. There is no hardware cost to this, these declarations are simply declarations to "view" the same bits differently (as 2's-complement coded integers). The difference arises later, when we apply certain operators to these values. For example, lines 24-25 compute the SLT (Set Less Than (signed)) and SLTU (Set Less Than Unsigned) operations. SLT uses the signed values `iv1` and `iv2`, whereas SLTU uses the unsigned values `v1` and v̌2. Between the *bsc* compiler and the Verilog back-end, different code will be generated for the "`<`" operator to perform the correct kind of comparison.

Line 18 extracts a 5-bit "shift amount" from the `rs2` value for the shift operators SLL, SRL and SRA. Line 39 extract a 5-bit "shift amount" from the `imm` value for the shift operators SLLI, SRLI and SRAI.

SRL (Shift Right Logical) and SRA (Shift Right Arithmetic) differ in whether they treat the argument as a signed or unsigned value, the difference being whether the new bits shifted into the most-significant bit are zero (SRL) or replicate the most-significant bit (SRA). SRLI and SRAI exhibit a similar difference.

In lines 28 (SRA) and 49 (SRAI) we finally apply the "`pack`" operator to produce the result. This is because the expression "`(iv1 >> shamt)`" has type `Int#(XLEN)` whereas the result needs to be of type `Bit#(XLEN)`. The "`pack`" operator performs this type-change for us.

Lines 19-32 define the `y_OP` result when the opcode is `opcode_OP` *i.e.,* `7'b_011_0011`, *i.e.,* the "3-address" operators where the inputs come from `rs1` and `rs2`. `y_OP` defaults to 0 when it is not an `op_OP`.

Lines 40-49 define the `y_OP_IMM` result when the opcode is `opcode_OP_IMM` *i.e.,* `7'b_001_0011`, *i.e.,* the "2-address" operators where one input come from `rs1` and the other input comes from an immediate value in the instruction. `y_OP_IMM` defaults to 0 when it is not an `op_OP_IMM`.

Finally, line 45 combines these results using the "OR" function. We rely on the fact that exactly one of `y_OP` and `y_OP_IMM` can be relevant; the other one must be zero (and therefore has no effect through the OR'ing).

---

**Exercise 6.8:**

Lines 11-45 could instead have been written this way:

```
1    Bit #(XLEN) y = 0;
2    if (opcode == opcode_OP) begin
3       ...
4       ... y = ...
5    end
6    else if (opcode == opcode_OP_IMM) begin
7       ...
8       ... y = ...
9    end
10   return y;
```

Discuss the hardware tradeoffs between writing it in these two ways. *H*ints: Consider:

- Sequentiality of if-then-else.
- Ability (or not) to prove exhaustiveness of conditions in nested if-then-else.
- Ability (or not) to prove mutual-exclusivity of conditions in nested if-then-else.
- Discussion in Section 4.11.1 on parallel and sequential multiplexers (mux). Note: in our code, we have explicitly coded a parallel mux.

**Exercise 6.9:**

Note that the ISA has ADD and ADDI instructions, but no corresponding SUB and SUBI (subtract) instructions. Why not?

**Exercise 6.10:**

Justify the presence or absence of the "`pack`" operator in each case of `fn_IALU`.

**Exercise 6.11:**

Suppose we want to extend `fn_IALU` so it also works when XLEN=64 (*i.e.,* for RV64I). What needs to change to accommodate this?

*Hint:* it only matters in the shift-amount of the shift instructions, where the shift-amount can be 6-bits wide instead of 5-bits (allowing a maximum of 63-bit shifts instead of 31 bits).

□

---

## 6.7 No separate functions for Execute DMem and Retire

We don't need any separate functions for the Execute DMem step in Figure 6.1 because `fn_Dispatch` has created a `Mem_Req` struct that we can send directly to memory. Similarly, the memory returns a `Mem_Rsp` that is sent directly to the Retire step.

The code for the Retire step is discussed in detail separatly in Chapter 10 for Drum and Chapter 13 for Fife. Although functionally the same, the codes are structurally sufficiently different that it is not worth attempting to abstract any common functionality into a shared function.

# Chapter 7

# BSV: Modules and Interfaces: Registers, Register Files and FIFOs

## 7.1 Introduction

It is good engineering practice to organize the code for any non-trivial system, whether in hardware or software, into a well-structured composition of smaller, manageable *modules*. Each module should have a clear, independent specification so that it can be understood on its own, and so that it can transparently be substituted by another module with the same functionality but perhaps other desirable properties (*e.g.,* speed, area, power). The external specification of a module—its "interface"—should not rely on, and ideally not even mention, internal implementation details of the module. For example, each of the units shown in Figure 3.1 could be a separate module.

This chapter is mostly a BSV chapter; we discuss modules and interfaces in BSV, including a few that are provided by the BSV libraries and that we will use in subsequent chapters to implement our RISC-V CPUs.

## 7.2 Modules: state, interfaces and behavior

Modules encapsulate modifiable *state*. Examples include Registers, Register Files and FIFOs (all of which are discussed later in this chapter). Modules with state are the only entities containing values that *persist* over time, *i.e.,* a value "written" at one moment in time can be "read" at a later moment in time.

Modules also encapsulate external behavior, using *interface methods*. In this sense they are similar to "objects" in object-oriented programming languages such as C++, Java, and Python. A BSV module is like an object constructor; a module *instance* is like an object; its internal state is like the internal, private "members" of the object, and its interface is *a set of methods* that can be invoked like functions/procedures, and which can access the internal state.

Modules and interfaces clearly separate concerns of externally-visible functionality ("external API"; *what* a module does; a module *specification*) *vs.* internal implementation details (*how* the module does it).

BSV modules are typically organized in a *hierarchy*—a top-level module, which instantiates sub-modules which, in turn, instantiate lower-level modules, and so on. In Drum and Fife:

Top-level module
    CPU module
        CPU sub-modules
            Library modules: Registers, Register file, FIFOs, ...
    Memory system
        Memory module(s)
        MMIO device modules (*e.g.,* UART, timer, interrupt controller)

In the next several sections we describe the concepts of BSV modules and interfaces. These sections may require re-reading a couple of times; the concepts become properly internalized only after seeing/using/creating several examples.

NOTE: We sometimes write BSV modules that do not themselves contain internal state, for stylistic and readability reasons. One example is seen in Section 7.5.6 where a module is used to encapsulate the logic of connecting two complementary interfaces.

### 7.2.1 Internal behavior (*rules*)

Unlike most programming languages, BSV modules typically also contain *internal* free-running processes called *rules* that run concurrently with the rest of the system (all other rules in the system). Rules realize the independent, concurrent, internal behavior of a module.

Rules are discussed in more detail in Chapter 14. Before that, in Chapter 9 we will discuss BSV's special notation for FSMs (Finite State Machines), which are simpler to use as a first step.

### 7.2.2 Interface declarations

An *interface declaration* in BSV declares a new interface, which is a new BSV *type*, and looks like this:

```
interface interface-type;
    ... method and sub-interface declarations ...
endinterface
```

The interface represents the external view of a module, *i.e.,* it declares a set of *methods* that can be invoked from an external context. Each method declaration only lists its arguments and their types, and the method's overall result type. The *body* of the method is defined in the module declaration of each module that offers this interface type.

Interfaces can be nested (can contain sub-interfaces which themselves have methods or sub-sub-interfaces, and so on). This is just a syntactic abstraction mechanism; ultimately, all

interactions with a module are through its methods, whether at the top level of the interface type, in a sub-interface, in a sub-sub-interface, *etc.*

There can be many module declarations each of which offers the same interface, *i.e.,* these are different *implementations* of the same interface. For example, the BSV library contains a repertoire of FIFO modules, all of which have the same FIFO interface type. Different implementations of a particular interface type typically differ on some dimension such as performance (latency, bandwdith, MHz), silicon area/FPGA gates, power consumption, *etc.*. One chooses a particular implementation based on such practical requirements.

Sections 7.3.1, 7.4.1 and 7.5.1 show the interface declaration for BSV library modules: Registers, Register Files and FIFOs, respectively.

### 7.2.3 Module declarations

A *module declaration* in BSV describes a module with a particular interface type:

```
(* synthesize *)
module module-name ( interface-type );
    ... instantiation of module state (registers, FIFOs, other sub-modules) ...
    ... behavior (rules and FSMs) ...
    ... interface (API) method implementations ...
endmodule
```

The `(* synthesize *)` attribute at the top is optional. With this attribute, the *bsc* compiler create a separate Verilog module for this BSV module. Without the attribute, the compiler will "inline" it into any parent module where it is instantiated.

A module declaration declares a module *constructor*. The constructor can be invoked multiple times to obtain multiple *instances* of the module.

NOTE:

> A notable difference between BSV and other HDLs (Verilog, SystemVerilog and VHDL) is that even a lowly register is not special; it is just another module, with an interface containing "`_read()`" and "`_write()`" methods.
>
> In fact BSV treats *all* "state elements" (components that store persistent values) uniformly as modules with interfaces.

### 7.2.4 Module instantiation and method invocation

A module is *instantiated* using this syntax:

*interface-type x* `<-` *constructor* `(`*constructor-arg*,...,*constructor-arg*`);`

This creates a new instance of the module and binds the offered interface to the identifier *x*. Some constructors have no arguments, in which case even the parentheses surrounding the arguments can be omitted.

Subsequently, methods of the module can be invoked using the syntax

*x.method-name* `(`*method-arg*,...,*method-arg*`);`

Some methods have no arguments, in which case even the parentheses surrounding the arguments can be omitted.

## 7.3 BSV Library Modules: Registers

A register is the simplest storage element in digital hardware, a single memory cell containing a single value (represented as a bit-vector). We can (over-)write with a new value, and we can read out the value stored by the most recent write.

### 7.3.1 `Reg#(t)`, the register interface from the BSV library

The standard register interface type in BSV has two methods:

```
1 interface Reg #(t);
2    method t _read();
3    method Action _write (t x);
4 endinterface
```

Here, "`t`" is the type of value stored in the register (discussed in more detail below).

The `_read()` method (with no arguments) just returns the value stored in the register, of type `t`. The `_write()` method takes one argument, a value of type `t`, and stores it in the register, over-writing any previous value and holding the new value until over-written by the next `_write()`.

We will explain the `Action` type of the `_write` method in more detail later. For now, just think of it as the type of any method that is a pure side-effect, *i.e.,* the method modifies some internal state of the module, and does not return any value.

### 7.3.2 Registers are strongly typed

Unlike Verilog, SystemVerilog and VHDL, BSV registers are "strongly typed". Each register instance can only hold values of one particular type, specified at the place where the register is instantiated.

Further, the register-contents type need not be `Bits#()`; it can more generally be *any* BSV type that has a representation in bits. Thus, the type of a value in a register can be an enum, a struct, a nested struct, *etc.*, if we have used a `deriving(Bits)` declaration (or its explicit analog) to ensure that it has a representation in bits.

Any attempt to read or write a value into a register that does not match the declared type will provoke a compile-time type-checking error from the *bsc* compiler.

### 7.3.3 mkReg(*v*), a register module (consructor) from the BSV library

A standard BSV library register module is `mkReg`. It is used to instantiate a new register, with a specified reset value, using a statement like this:

```
1   Reg #(Bit #(XLEN)) rg_pc <- mkReg (0);
```

Here we declare a new identifier `pc` with interface type `Reg#(Bit#(XLEN))` (the register interface type) and bind it to the interface offered by a newly instantiated register. The "0" argument to `mkReg()` specifies the reset-value of the register, *i.e.,* the value held in the register immediately after the hardware has been reset.

An alternative register constructor provided by the BSV library is `mkRegU`, where the "U" indicates that it is uninitialized, *i.e.,* has no specified reset value:

```
1   Reg #(Bit #(XLEN)) rg_pc <- mkRegU;
```

`mkRegU` instantiates a register with an unspecified (unpredictable) reset value, and hence does not need an argument.

### 7.3.4 Syntactic shorthands for register access

Registers are so ubuiquitous in digital design that BSV provides some special syntactic shorthands for reading and writing registers.

Just mentioning a register in an expression can be used as a shorthand for invoking its `_read` method. Thus, the expression:

```
rg_pc + 4
```

is shorthand for:

```
rg_pc._read + 4
```

To invoke the `_write` method on a register, one can use a conventional assignment statement. Thus, the expression:

```
rg_pc._write (v)
```

can be written like this:[1]

```
rg_pc <= v
```

[1] Rather than use "`=`" or "`:=`" common in software programming languages, we use "`<=`", which is the Verilog/SystemVerilog notation for "delayed assignment".

A statement like this:

```
rg_pc <= rg_pc + 4
```

contains both shorthands:

```
rg_pc._write (rg_pc._read + 4)
```

NOTE: The use of the "`rg_`" prefix in the above examples is just our own syntactic convention, and not required in BSV syntax, where any legal identifier can be bound to a register interface. We will be mixing identifiers bound to ordinary values and identifiers bound to register interfaces in various expressions. The `rg_` prefix reminds us that there is an implicit "`_read`" on the latter.

## 7.4 BSV Library Modules: Register files

A register file is an array of registers with a common pair of methods to read or write a particular register identified by an index, which is an argument to the methods for reading and writing.

### 7.4.1 The register file interface `RegFile#(index_t,data_t)` from the BSV library

The standard register file interface type in the BSV library is:

```
1 interface RegFile #(type index_t, type data_t);
2     method Action upd (index_t addr, data_t d);
3     method data_t sub (index_t addr);
4 endinterface: RegFile
```

Here, "`index_t`" is the type for the index, which we use to identify one of the registers in the register file. For RISC-V, since we have 32 registers, we will use `Bit#(5)` as the index type.

"`data_t`" is the type of value stored in each of the registers. For RISC-V, this will be `Bit#(XLEN)`.

The *rf*`.upd(j,v)` method allows us to store the value $v$ in the $j$'th register of register file *rf*. The *rf*`.sub(j)` method returns the current value $v$ in the $j$'th register of register file *rf*.

NOTE: The index type `index_t` can be any type that has a representation in bits, *i.e.,* for which we have used the `deriving(Bits)` annotation in the type declaration (or for which we have provided a so-called `Bits` instance explicitly).

BSV Register files, like BSV registers, are strongly typed. At time of instantiation of a register file *rf*, we specify its `index_t` and `data_t` types. In subsequent uses of *rf*, the provided index and data value, and returned data value, must have exactly those types (else the *bsc* compiler will raise a compile-time type-error.).

### 7.4.2 `mkRegFileFull`, a register file module (constructor) from the BSV library

The BSV library contains a couple of register file modules (constructors). For RISC-V we can use `mkRegFileFull`:

```
RegFile #(Bit #(5), Bit #(XLEN)) gprs <- mkRegFileFull;
```

Here we declare a new identifier `gprs` with interface type `RegFile#(Bit#(5),Bit#(XLEN))` (the register file interface type) and bind it to the interface offered by a newly instantiated register file. The number of registers in the register file is known from the full range of the index type `Bit#(5)`, *i.e.,* it will have 32 registers, indexed from 0 to 31. Each register is `XLEN` bits wide.

## 7.5 BSV Library Modules: FIFOs

FIFOs (First-in-First-out) elements are *ordered queues* of values and are broadly useful in many hardware designs (arguably as useful as registers). We can enqueue a new value into a FIFO at the tail (back) of the queue, and dequeue a value from the head (front) of the queue. Most BSV FIFOs are automatically "flow-controlled", *i.e.,* it is impossible to enqueue into a full FIFO and to dequeue from an empty FIFO.

### 7.5.1 `FIFOF#(t)`, the FIFO interface from the BSV library

A standard FIFO interface type in the BSV library is:[2].

```
1 interface FIFOF #(t);
2    method Bool notEmpty();
3    method Bool notFull();
4    method t first();
5    method Action deq();
6    method Action enq (t x);
7    method Action clear();
8 endinterface
```

Here, "`t`" is the type of values stored in the FIFO (discussed in more detail below).

The `f.notEmpty()` and `f.notFull()` are simple predicates to test if a FIFOF `f` is empty or full, respectively.

The `f.first()` and `f.deq()` methods are used to access the head of the queue. They are only available if the FIFO is not empty. The `first()` method returns the value at the head of the queue. This is non-destructive, *i.e.,* it does not modify the FIFO. The `f.deq()` method modifies the FIFO: it discards the value at head of the queue and advances the queue.

[2]The BSV library also defines the `FIFO#(t)` interface which is the same as the `FIFOF#(t)` except that it omits the `notEmpty` and `notFull` methods. We prefer the latter, which provides more flexibility.

The `f.enq(x)` method is used to access the tail of the queue, and is only available if the FIFO is not full. It modifies the FIFO by appending the argument `x` to the tail of the queue.

The `clear` method is used to empty the queue immediately (discard all its contents).

Notice that the `FIFOF#(t)` interface does not indicate the *capacity* of the FIFO, *i.e.,* the number of elements it can hold from head to tail. This is deliberate; we may choose different capacities for each FIFO instance as required by its use context. We also want to be able flexibly and transparently to substitute a FIFO with another that has greater or less capacity.

### 7.5.2 `mkFIFOF`, a FIFO module (constructor) from the BSV library

The BSV library contains many different FIFO modules (constructors): single-element FIFOs, FIFOs of a specified depth (queue length), FIFOs with and without automatic flow-control, *etc.* In Drum we use this one:

```
1   FIFOF #(Mem_Req) f_to_IMem   <- mkFIFOF;
2   FIFOF #(Mem_Rsp) f_from_IMem <- mkFIFOF;
```

Here we declare a new identifier `f_to_IMem` with interface type `FIFOF#(Mem_Req)` and bind it to the interface offered by a newly instantiated FIFO. Similarly, we declare a new identifier `f_from_IMem` with interface type `FIFOF#(Mem_Rsp)` and bind it to the interface offered by a newly instantiated FIFO. Due to BSV's strong-typing, the first FIFO can only hold items of type `Mem_Req` and the second FIFO can only hold items of type `Mem_Rsp`.

Different module-constructors may or may not have arguments. This example from Fife uses a different BSV library FIFO constructor:

```
1   FIFOF #(RR_to_Retire)  f_RR_to_Retire <- mkSizedFIFOF (8);
```

This instantiates a FIFO whose queue capacity is 8. Note that module constructor arguments can play different roles. In `mkReg` above, the argument (0) became a dynamic value, the value held by the register after reset. Here, the argument (8) only describes *structure*, *i.e.,* the size of the FIFO.

`mkFIFOF` happens to have capacity 2, although it will support sustained simultaneous enqueueing and dequeuing only when its average occupancy is $\leq 1$ (zero or one element in the queue).

All these FIFO modules are initially empty (containing zero items) on reset, *i.e.,* at the start of time for the circuit.

### 7.5.3 FIFOs are strongly typed

Each BSV FIFO instance can only hold values of one particular type.

Further, the FIFO-contents type need not be `Bits#()`; it can more generally be *any* BSV type that has a representation in bits. Thus, the type of values in a FIFO can be an enum, a struct, a nested struct, *etc.*, if we have used a `deriving(Bits)` declaration (or its explicit analog) to ensure that it has a representation in bits.

### 7.5.4 Semi-FIFO interfaces for each end of a FIFO

FIFOs are often used to connect two separate modules together, for one module to communicate values to the next one. For example, the Fetch step communicates memory requests to memory. In this situation, one module only interacts with the "enqueue" side, and the other module only interacts with the "'dequeue" side.

In these situations we will also find it useful to use the following "Semi-FIFO" interfaces interfaces for each "end" of a FIFO queue:

```
interface FIFOF_O #(t);
   method Bool notEmpty();
   method t first();
   method Action deq();
endinterface
```

```
interface FIFOF_I #(t);
   method Bool notFull();
   method Action enq (t x);
endinterface
```

There is no extra hardware implied here; these are simply limited "views", or abstractions of an existing FIFO interface.

### 7.5.5 Interface-transformer functions

The idea of "viewing" the output-side of a `FIFOF` interface as a `FIFOF_O` interface can be expressed in a BSV function:

```
function FIFO_O #(t) to_FIFOF_O (FIFOF #(t) f);
   interface FIFOF_O #(Mem_Req) fo_IMem_req;
      method Bool notEmpty();
         return f.notEmpty;
      endmethod

      method t first();
         return f.first;
      endmethod

      method Action deq();
         f.deq;
      endmethod
   endinterface
endinterface
```

---

**Exercise 7.1:**

Write a similar function to transform a `FIFOF` interface into a `FIFOF_I` interface, with `notFull` and `enq` methods.

□

---

### 7.5.6 Connecting FIFOs

We will frequently want to connect the output of one FIFO to the input of another FIFO. For example, in Fife, the interface of the Fetch stage includes this semi-FIFO sub-interface to commicate `F_to_D` values to the Decode unit:

```
1  interface F_IFC;
2     ...
3     interface FIFOF_O #(F_to_D)  fo_F_to_D;
4     ...
5  endinterface
```

The interface of the Decode stage has this corresponding sub-interface to receive those values:

```
1  interface D_IFC;
2     ...
3     interface FIFOF_I #(F_to_D)  fi_F_to_D;
4     ...
5  endinterface
```

The the CPU module, at the next level up, instantiates the Fetch and Decode stages. Then, they can be connected with a simple `mkConnection` one-liner:

```
1  module mkCPU (CPU_IFC);
2     ...
3     // Instantiate Fetch and Decode stages
4     F_IFC  stage_F  <- mkF;
5     D_IFC  stage_D  <- mkD;
6     ...
7     // Connect the F_to_D flow
8     mkConnection_O_to_I (stage_F.fo_F_to_D, stage_D.fi_F_to_D);
9     ...
10 endmodule
```

There is actually no magic in this! First, `mkConnection_O_to_I` is just another BSV module which happens to have an "empty" interface (called `Empty`, with no interface methods), so line 8 is actually shorthand for another module instantiation (the shorthand omits the "`Empty tmp <-`" left-hand side):

```
// Connect the F_to_D flow
Empty tmp <- mkConnection_O_to_I (stage_F.fo_F_to_D, stage_D.fi_F_to_D);
```

Then, the module `mkConnection_O_to_I` can be written simply in BSV itself:

```
1 module mkConnection_O_to_I #(FIFOF_O #(F_to_D) f,    // module argument
2                              FIFOF_I #(F_to_D) d)    // module argument
3                             (Empty);                 // module interface
4    rule rl_connect;
5        let x = f.first;
6        f.deq;
7        d.enq (x);
8    endrule
9 endmodule
```

`mkConnection_O_to_I` is a module with two arguments `f` and `d` and producing an empty interface. In BSV, Verilog and SystemVerilog syntax, a module's arguments are provided in `#(...)` and its interface follows in `(...)`.

The module contains a *rule*, which is an infinite process. It binds `x` to `f.first`, the head of the `f` queue, and discards it from the queue (`f.deq`). It enqueues the value `x` into `d`. Being an infinite process, it repeats this every time this is possible.

Because of the automatic flow-control in BSV FIFOs, this rule will only execute when `f` is non-empty (contains an item, available to dequeue) and `d` is not full (has space, available to enqueue).

NOTE:

**Advanced BSV topic:** What if we want to connect the two semi-FIFOF interfaces with the arguments in the opposite order, *i.e.,* a `FIFOF_I` interface to a `FIFOF_O` interface? We could write a corresponding `mkConnection_I_to_O` module. What if we want to connect an ARM AXI4 M interface to an ARM AXI4 S interface? We could write a corresponding `mkConnection_AXI4_M_to_S` module.

When there are many different kinds of connection, inventing new module names `mkConnecton_X_to_Y` for each pair of interface types `X` and `Y` becomes tedious.

BSV contains a mechanism called "Typeclasses" and "Typeclass instances" that allows us to reuse the name `mkConnection` for the connection module for every such pair of interface types.

In Programming Language design this issue and solutions are called "overloading".

### 7.5.7 `mkPipelineFIFOF` and `mkBypassFIFOF`: constructors from the BSV library

We mention two more FIFO module constructors from the BSV library— `mkPipelineFIFOF` and `mkBypassFIFOF`—because they are used heavily in Fife code shown in Chapter 13.

To first approximation, they can be considered merely as substitutes for `mkFIFOF`. They have subtle *performance* differences from `mkFIFOF`, which is why we use them in Fife. We will discuss these performance properties in more detail in Chapter 14.

## 7.6 Polymorphic and Monomorphic Types

The previous sections showed several *polymorphic* types:

- `Reg #(t)`
- `RegFile #(ix_width, reg_width)`
- `RISCV_GPRs_IFC #(reg_width)`
- `FIFOF #(t)`
- `FIFOF_I #(t)`
- `FIFOF_O #(t)`

In each case, there is a *type constructor* (`Reg`, `RegFile`, ..., `FIFOF_O`) applied to some arguments (`t`, `ix_width`, `reg_width`). The general syntax is:

*type-constructor* `#(`*arg*, ..., *arg*`)`

(When a type constructor has no arguments, we can omit "`#()`".)

When the argument of a type-constructor is an identifiers beginning with a lower-case letter, this indicates that this is a *type variable*, *i.e.,* it is place-holders for some specific type that will be instantiated later/elsewhere.

A polymorphic type (a type containing one or more type-variables) represents all possible types one can obtain by instantiating the type variables with specific, concrete types. A type with no type-variables is also called *monomorphic*.

Note that it is not just library types (`Reg`, `RegFile`, `FIFOF`) that may be polymorphic. In the previous sections, we defined new types `RISCV_GPRs_IFC #(reg_width)`, `FIFOF_I #(t)` and `FIFOF_O #(t)` that are also polymorphic.

---

**Exercise 7.2:**

The types `Mem_Req` and `Mem_Rsp` (Sections 5.3.2 and 5.3.4) are monomorphic. Write polymorphic versions of these types that are parameterized on `xlen`.

□

---

### 7.6.1 Polymorphic Modules and Synthesizability into Verilog

In Section 7.2.3 we mentioned without discussion that the "`(* synthesize *)`" attribute preceding a `module` declaration controls whether the *bsc* compiler generates a Verilog module for this BSV module or whether it is inlined into its parent module wherever it is instantiated.

Not all BSV modules can be compiled one-to-one into Verilog modules. Broadly speaking, polymorphic modules cannot be separately compiled into Verilog modules. The reason is that polymorphism in BSV is very powerful and, beyond the expressive power of Verilog.

This does not mean that we cannot use polymorphic modules in a BSV design; of course we can! It just means that, at each instance of the module where we have instantiated it and fully specified the types ("monomorphized" the types), the *bsc* compiler at that place has enough information to generate Verilog for that instance.

For example, consider the polymorphic `mkRISCV_GPRs` module Section 8.2. We can directly instantiate this module in a CPU module:

```
module mkCPU;
   ...
   RISCV_GPRs_IFC #(XLEN) gprs <- mkRISCV_GPRs;
   ...
endmoduile
```

At this instantiation position, the *bsc* compiler knows the concrete value (`XLEN`) of the type-variable `xlen`, and so can generate Verilog code for this `mkRISCV_GPRs` instance. In the final Verilog code, there will not be any separately identifiable Verilog code for the register file, it would just be part of the `mkCPU` module Verilog.

If we really wanted to generate a separately identifiable Verilog module for the register file, we can write a monomorphic wrapper module:

```
(* synthesize *)
module mkRISCV_GPRs_V (RISCV_GPRs_IFC #(XLEN));
   RISCV_GPRs_IFC #(XLEN) ifc <- mkRISCV_GPRs;
   return ifc;
endmoduile
```

This module `mkRISCV_GPRs_V` is monomorphic because the type-variable `xlen` has been instantiated to the concrete type `XLEN`, and so the "`(* synthesize *)`" attribute will be honored by the *bsc* compiler to produce a corresponding Verilog module.

We can then replace our earlier instantiation of `mkRISCV_GPRs` in the CPU module with this monomorphic module:

```
module mkCPU;
   ...
   RISCV_GPRs_IFC #(XLEN) gprs <- mkRISCV_GPRs_V;
   ...
endmoduile
```

---

**Exercise 7.3:**

In `mkRISCV_GPRs_V` replace the explicit type declaration of `ifc` with the `let` keyword.

**Exercise 7.4:**

Write a monomorphic wrapper for the BSV library `mkFIFOF` module that specializes it into a FIFO that only carries `Bool` values. Add an annotation so the wrapper becomes a separate Verilog module. Compile it and study the generated Verilog.

□

---

# Chapter 8

# RISC-V: Modules for GPRs and CSRs

## 8.1 Introduction

In this chapter we discuss modules for the RISC-V GPRs (General Purpose Registers) and CSRs (Control and Status Registers).

## 8.2 A register file for RISC-V GPRs, with special treatment of `x0`

In RISC-V, register `x0` (index 0) is defined as "always zero". Any value written to `x0` is ignored/discarded, and any read from `x0` always returns 0. So, presumably, we do not need an actual register to hold this value, just some circuitry to ensure that we always get 0 when we try to "read" from `x0`.

In the previous section, we used the module `mkRegFileFull` to instantiate a register file with 32 registers (inferring 32 from the full range of the index type `Bit#(5)`). Instead, we could use an alternate register file module from the BSV library that allows us to provide, as module constructor arguments, the lower and upper indexes of interest. This instantiates exactly 31 registers indexed from 1 to 31, thereby saving XLEN bits of register state in our hardware.

```
RegFile #(Bit #(5), Bit #(XLEN)) gprs <- mkRegFile (1, 31);
```

Regardless of whether we instantiated 31 or 32 registers, RISC-V instructions can (and do) use `x0` as a source or destination register, so we need circuitry to deal with attempts to read/write `x0`. One possible solution is to make a "wrapper" module `mkRISCV_GPRs` around the library register file module.

Although we could have reused the `RegFile #(t1,t2)` interface, we take the opportunity to define a new interface `RISCV_GPRs_IFC` that has some RISC-V specific method and argument names, for reading the rs1, rs2 values (register source 1 and 2) and writing the rd value (register destination):

src_Common/RISCV_GPRs.bsv: line 25 ...

```
1 interface RISCV_GPRs_IFC #(numeric type xlen);
2    method Bit #(xlen) read_rs1 (Bit #(5) rs1);
```

```
3     method Bit #(xlen) read_rs2 (Bit #(5) rs2);
4     method Action      write_rd (Bit #(5) rd, Bit #(xlen) rd_val);
5  endinterface
```

Here is the module implementing the interface:

src_Common/RISCV_GPRs.bsv: line 41 ...

```
1  module mkRISCV_GPRs (RISCV_GPRs_IFC #(xlen));
2     RegFile #(Bit #(5), Bit #(xlen)) rf <- mkRegFileFull;
3
4     method Bit #(xlen) read_rs1 (Bit #(5) rs1);
5        return ((rs1 == 0) ? 0 : rf.sub (rs1));
6     endmethod
7
8     method Bit #(xlen) read_rs2 (Bit #(5) rs2);
9        return ((rs2 == 0) ? 0 : rf.sub (rs2));
10    endmethod
11
12    method Action write_rd (Bit #(5) rd, Bit #(xlen) rd_val);
13       rf.upd (rd, rd_val);
14    endmethod
15 endmodule
```

The module instantiates a library register file `rf`. The methods simply invoke the underlying `rf` methods. The read-methods override this by returning 0 when the index is 0.

---

**Exercise 8.1:** In `mkRISCV_GPRs` we write the value when $j$ is zero, but we ignore it on reads. Write a variant where, in `write_rd`, we alway write 0 when the index `rd` is zero, and the read methods no longer check if `rs1` or `rs2` are 0.

In this variant, what happens if we try to read `x0` before it is written?

Compare the circuitry generated in the original and in the variant. Why might we choose one over the other?

**Exercise 8.2:** In `mkRISCV_GPRs` suppose we use `mkRegFile(1,31)` instead of `mkRegFileFull`. What needs to change to accommodate this?

□

---

### 8.2.1 Inlined *vs.* separate module `mkRISCV_GPRs`

The above interface and module definitions were parameterized with "`xlen`" which is a type-variable (starting with a lower-case letter). The interface and module are thus polymorphic in the width of the data stored in the registers. Thus, this module can be instantiated in RV32 and RV64 designs, instantiating `xlen` to 32 and 64, respectively.

Being polymorphic, it will also be inlined wherever it is instantiated. If you look at the generated Verilog, there will be no trace of `mkRISCV_GPRs`; the module code will have been integrated (inlined) into the parent module's code.

A useful trick is to write a thin, non-polymorphic wrapper for the module; being non-polymorphic, it can be compiled without inlining into a distinct Verilog module:

src_Common/RISCV_GPRs.bsv: line 60 ...

```
(* synthesize *)
module mkRISCV_GPRs_synth (RISCV_GPRs_IFC #(XLEN));
   let ifc <- mkRISCV_GPRs;
   return ifc;
endmodule
```

Here, we have instantiated the module using `XLEN` which is not a type-variable, it is defined specifically as 32 or 64 (see Sec 4.12.4). The `(*synthesize*)` attribute can now be respected by the *bsc* compiler and it will produce a Verilog module `mkRISCV_GPRs_synth`.

## 8.3 A register file for RISC-V CSRs

The RISC-V CSRs (Control and Status Registers) are not "just another register file". Here are some significant differences:

- There are 32 GPRs, addressed with a 5-bit index. All of them (except one, `x0`) are used in programs. We can thus use a *dense*, packed implementation (`RegFile` from the BSV library).

  CSRs are addressed with a 12-bit index (CSR address). But in our implementation we will use just a handful of CSRs (`mtvec`, `mtepc`, `mcause`, `mtval` and a few more), with non-consecutive addresses, *i.e.,* the addresses are *sparse*; many (most) 12-bit address values are unused.

- Each GPR (except for `x0`) is "memory-like". When a value is written, it remains available for all subsequent reads until the next write. All those reads return the same value—the value most recently written.

  CSR reads can, in general, have side effects. CSR writes, in addition to writing a value, can have other side effects as well. A CSR read may not return the same value as the value most recently written.

For these reasons, we implement each CSR with a separate, ordinary register. Here are the CSRs we need for exception-handling:

src_Common/CSRs.bsv: line 56 ...

```
   Reg #(Bit #(XLEN)) csr_mtvec   <- mkReg (0);
   Reg #(Bit #(XLEN)) csr_mepc    <- mkReg (0);
   Reg #(Bit #(XLEN)) csr_mcause  <- mkReg (0);
   Reg #(Bit #(XLEN)) csr_mtval   <- mkReg (0);
```

Here are the definitions of standard RISC-V 12-bit addresses for these CSRs:

src_Common/CSR_Bits.bsv: line 23 ...

```
Bit #(12) csr_addr_MTVEC     = 'h305;
Bit #(12) csr_addr_MEPC      = 'h341;
Bit #(12) csr_addr_MCAUSE    = 'h342;
Bit #(12) csr_addr_MTVAL     = 'h343;
```

To write a CSR we use a case statement that selects on the CSR address and writes to the particular CSR.

src_Common/CSRs.bsv: line 64 ...

```
function ActionValue #(Bool)
         fav_csr_write (Bit #(12) csr_addr, Bit #(XLEN) csr_val);
   actionvalue
      ...
      Bool exception = False;
      case (csr_addr)
         ...
         csr_addr_MTVEC:   csr_mtvec   <= csr_val;
         csr_addr_MEPC:    csr_mepc    <= csr_val;
         csr_addr_MCAUSE:  csr_mcause  <= csr_val;
         csr_addr_MTVAL:   csr_mtval   <= csr_val;
         default:          exception = True;
      endcase
      return exception;
   endactionvalue
endfunction
```

This function returns boolean True on success, and False on failure. For the moment, failure only means that the argument CSR address was bad, *i.e.,* it referred to some unknown CSR. Failure can also occur if we try to write to a "read-only" CSR (we will see an example later, CSR TIME).

Similarly, to read a CSR we use a case statement that selects on the CSR address and reads the particular CSR.

src_Common/CSRs.bsv: line 89 ...

```
function ActionValue #(Tuple2 #(Bool, Bit #(XLEN)))
         fav_csr_read (Bit #(12) csr_addr);
   actionvalue
      ...
      Bool        exception = False;
      Bit #(XLEN) y         = ?;
      case (csr_addr)
         ...
         csr_addr_MTVEC:     y = csr_mtvec;
         csr_addr_MEPC:      y = csr_mepc;
         csr_addr_MCAUSE:    y = csr_mcause;
         csr_addr_MTVAL:     y = csr_mtval;
         default:            exception = True;
      endcase
      ...
      return tuple2 (exception, y);
   endactionvalue
endfunction
```

This function returns a pair of values (a 2-tuple). The first component, like the CSR-write function, is a boolean, True on success and False on failure. The second component is the value read from the CSR.

For the CSRs module we do not export the above read and write functions directly; they are used internally inside the module. The inteface declaration looks like this:

src_Common/CSRs.bsv: line 29 ...

```
interface CSRs_IFC;
   method Action init (Initial_Params initial_params);

   // CSRRXX instruction execution
   // Returns (True, ?) if exception else (False, rd_val)
   method ActionValue #(Tuple2 #(Bool, Bit #(XLEN)))
          mav_csrrxx (Bit #(32) instr, Bit #(XLEN) rs1_val);

   // Trap actions
   // Returns PC from MTVEC for trap handler
   method ActionValue #(Bit #(XLEN))
          mav_exception (Bit #(XLEN) epc,
                         Bool        is_interrupt,
                         Bit #(4)    cause,
                         Bit #(XLEN) tval);

   method Bit #(XLEN) read_epc;
endinterface
```

We will discuss these interface methods in more detail when we discuss trap-handling in Chapter 10. Suffice it to say, for now, that:

- the `mav_csrrx` method directly implements the CSRRxx instructions;
- the `mav_exception` method directly implements the CSR reads and writes needed when taking a trap, and
- the `read_epc` method directly reads the MEPC CSR as needed by the MRET instruction.

The former was described in Section 2.7.2, and the latter two were described in Section 2.7.

// ****************************************************************

# Chapter 9

# BSV: FSMs

## 9.1 Introduction

So far, we have only been discussing pure combinational functions, for which there is no concept of time. Combinational functions are just pure mathematical functions, "instantaneously" transforming input values to output values. However, a CPU, as shown in Figure 9.1 represents a *processs*, a behavior that evolves over time. For example the Drum

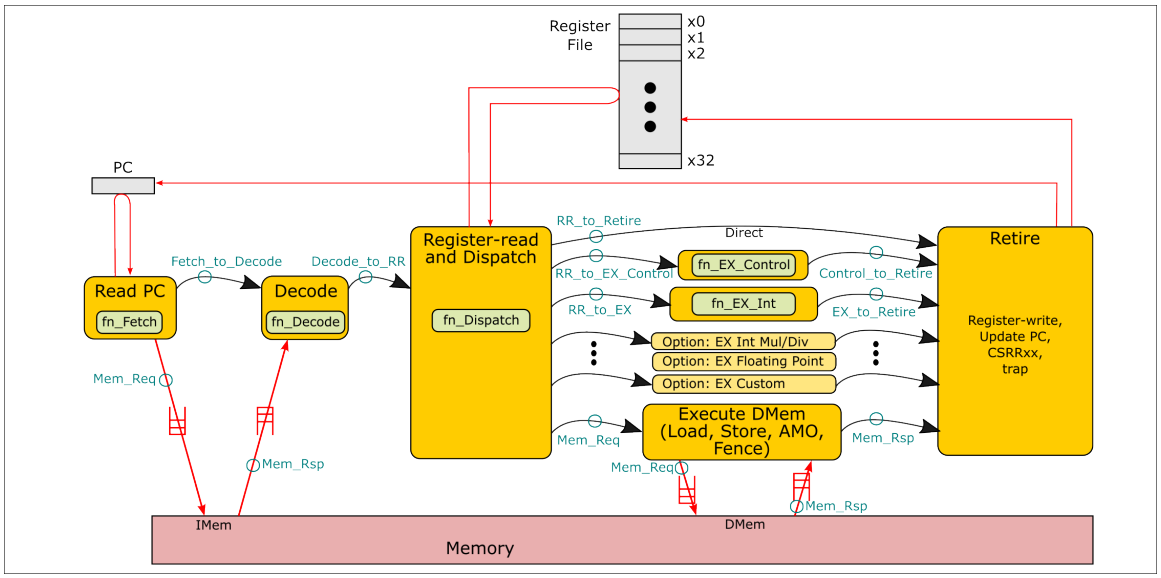



Figure 9.1: Simple interpretation of RISC-V instructions (same as Fig. **??**)

CPU executes one full instruction after another, repeating forever the flow along the black arrows in the diagram. For each instruction, first it performs a Fetch operation, which sends a request to memory. Some time later, the memory sends back a response, which is then processed by the Decode step, Register-Read-and-Dispatch step and then one of the Execute steps. The Execute Memory Ops step sends a request to memory. Some time later, the memory sends back a response, which is processed in the Retire step. Finally, it it loops back to the Fetch step, and the process repeats for the next instruction.

The simplest temporal process in hardware systems, perhaps the most classical, is the *FSM* (Finite State Machine). A classical notation for an FSM are so-called "bubble-and-arrow" diagrams: each bubble represents a distinct *state* of the system, and the arrows connecting bubbles represent *transitions* between states. Transitions are typically enabled by some predicate condition on the current state, and/or availability of some particular input from the environment. A transition moves the system to another state, and may output something as well to the environment. In bubble-and-arrow diagrams, each arrow is often labeled with the condition that enables the transition and any outputs produced by the transition.

Figure 9.1 can be interpreted as a bubble-and-arrow FSM diagram: each yellow rectangle (bubble) is a state, and the process transitions from state to state, thereby executing RISC-V instructions. This is exactly what the Drum CPU does. Arros labels in our figure not conditions and actions, rather the information (a struct) that is produced by one state and consumed by another.

In this chapter we describe some special constructs for FSMs in BSV. In the following chapters we will discuss using these constructs to code the Drum CPU.

NOTE: In the literature one may read about FSMs where, from a state, we could have a choice of transitions to different destination states. These are called *non-deterministic* FSMs. In our designs we are only concerned with deterministic FSMs where the conditions alway identify a unique possible next state.

### 9.1.1 Sequential FSMs, Concurrent FSMs, and Digital Hardware

Classical FSMs in the literature are *sequential* FSMs—every transition is from the current state to a unique, particular next state.[1]

Any digital system (including an entire computer) can theoretically be viewed as a single (possibly giant!) FSM. The current-state is the current collective state of every register bit and every memory bit in the system. State transitions depend on the current state and any external inputs. Although theoretically correct, this is an impractical, non-scalable, and not very useful way of viewing complex digital systems.

Most non-trivial digital hardware systems are better viewed as composed of multiple *concurrent, communicating FSMs*, *i.e.,* multiple classical FSMs running concurrently and independently and communicating with each other (an output from one FSM may be an input to another FSM). Different BSV module instances are separate FSMs, each running their own process(es). These separate FSMs may communicate with each other *via* shared state (registers, fifos, register files, *etc.*). This is a more *modular* and *scalable* way to think about complex digital systems.

Even though Drum CPU execution is a sequential FSM, the overall system can be viewed as a pair of concurrent FSMs: Drum and the Memory System. Each has its own internal FSM and behavior, and they communicate memory requests and responses back and forth.

For Fife, we will interpret *each* yellow box in Figure 9.1 as its own FSM. They all run concurrently (and concurrently with the memory system), and communicate various struct values (labels in Figure 9.1) between the FSMs.

[1] Even in non-deterministic FSMs, though there may be several possible next-states, exactly one next-state is non-deterministically chosen.

## 9.2 Rules and StmtFSM in BSV

The fundamental behavioral/temporal primitive in BSV is the *rule* but we will postpone a detailed discussion of rules until Chapter 14. BSV's `StmtFSM` is a higher-level notation that captures the idea of certain *structured* processes. `StmtFSM` is ultimately implemented (by the *bsc* compiler) with rules, so it does not add any fundamental semantic power to the language.

HISTORICAL NOTE:

In a software program execution, the fundamental temporal "flow" primitive is from one instruction to the next, or a BRANCH or JUMP. However, in most programming languages, we work at a much higher-level of abstraction, using using such linguistic structures as if-then-else, while-loops, for-loops, function calls and returns, *etc.*. Ultimately all these constructs are implemented (by a compiler) in terms of the primitives, instruction sequences/BRANCH/JUMP, and in that sense they do not add any fundamental new semantic power to programming. However, these structured constructs are useful for humans: for readability, for guiding our thinking, and for reasoning about the correctness of programs.

Early programmers, *e.g.,* using first versions of the Fortran programming language, worked directly with statement sequences (the analog of instruction sequences) and "GOTO" statements (the analog of BRANCH/JUMP), and their programs were a mess of control transfers without any structure, sometimes called "spaghetti code".

In the 1960s, Edsgar Dijkstra, a Dutch computer scientist (later to win the ACM Turing Award), recognized the importance of clear *structure* in composing programs. In 1968 he wrote a famous letter to the editor of the Communications of the ACM journal (Association for Computing Machinery) titled "Go To Statement Considered Harmful", a searing indictment of unstructured code.

The idea of constructing programs cleanly with nestable (composable) structures (if-then-else, while-loops *etc.*), so called "structured programming", grew out of that thinking; it is something we take for granted today.

`StmtFSM` is a sub-language within BSV for structured FSMs. It takes some basic, "instantaneous" actions, from which it builds sequential processes that sequence these actions using conditionals and loops.

We will use `StmtFSM` to code Drum, which is a simple sequential process. It will not be adequate to code Fife, which is a collection of concurrent FSMs; for that, we will use rules explicitly.

NOTE: We already briefly encountered a simple `StmtFSM`, with just a sequential block, in the testbench in Section 4.7.

## 9.3 Actions and the `Action` type

The fundamental building-block for `StmtFSM` is the "action", which is a statement/expression of type `Action`. Some common examples:

```
1   rg_pc <= rg_pc + 4;            // Assignment to a register
2   f_Fetch_to_Decode.deq;         // Dequeue a fifo
3   f_Decode_to_RR.enq (v);        // Enqueue into a fifo
4   $display ("Hello, World!");    // Print something (in simulation only)
```

We discussed the `Action` (and `ActionValue`) types in Section 4.6.1. We used them in the return type of the `fn_Fetch` function in Section 6.2. To recap, an expression with `Action` or `ActionValue` type is one that potentially has a side effect, as in each of the above example statements.

As discussed in Section 7.3.4 the first assignment statement is syntactic shorthand for:

```
1   rg_pc._write (rg_pc._read + 4)
```

*i.e.,* it is an invocation of the register `_write` method which, as described in Section 7.3.1 has type `Action`. Similarly, as described in Section 7.5.1, fifo `enq` and `deq` methods have return-type `Action`, so the statements `f_D_to_RR.enq (v)` and `f_D_to_RR.enq (v)` have type `Action`.

`$display()` is a built-in construct in BSV that also has type `Action`.

### 9.3.1 `Action` blocks: composing actions into larger actions

The `Action` type is recursive: it is either a primitive action (like those described just above), or it is a collection of things of type `Action`, collected using an `action` block (bracketed by the BSV keywords `action` and `endaction`). For example the above primitive actions can be collected into a single entity which itself has type `Action`:

```
1   action
2      rg_pc <= rg_pc + 4;          // Assignment to a register
3      f_F_to_D.deq;                // Dequeue a fifo
4      f_D_to_RR.enq (v);           // Enqueue into a fifo
5      $display ("Hello, World!");  // Print something (in simulation only)
6   endaction
```

Although the actions in an `action` block must be written in some textual order, `action` *blocks do not involve any sequencing.* There is no temporal ordering of the actions in an `action` block. All the actions in an `action` block (either directly in the block or, recursively in a sub-block) occur "instantly" and "simultaneously". In the example above, lines 2-5 could have been written in any order with no change in meaning/behavior.

### 9.3.2 Binding names in `Action` blocks

It is often convenient to give a meaningful name to a sub-expression in an `Action` block. For example:

```
1   action
2      Bit #(XLEN) next_pc = rg_pc + 4;
3      rg_pc <= next_pc;
4      $display ("Next PC is %08h", next_pc);
5   endaction
```

Here, we bind the identifier `next_pc` in line 2, and then use it in lines 3 and 4. We can often replace the type in the binding with the keyword `let`, if the type is obvious from the context:

```
1   action
2      let next_pc = rg_pc + 4;
3      rg_pc <= next_pc;
4      $display ("Next PC is %08h", next_pc);
5   endaction
```

The *scope* of the identifier, *i.e.,* the region of program text where it is available for use, is just the `Action` block (and inside any syntactically nested construct).

A binding, per se, *is not an action!*. It is just a convenience, giving a name to the value of the right-hand side expression.

Bindings (whether with a type or with `let`) impose some ordering on statements in the block: a binding of an identifier must precede any use of that identifier. In the previous two examples, line 2 (the binding) must precede lines 3 and 4 (the actions), but lines 3 and 4 could be written in the opposite order.

## 9.4 `StmtFSM`: sequences of actions

Our first construct that has temporal behavior is the `seq-endseq` block. Each item in the block is typically an entity of type |Action|, and they are performed sequentially, one after another.

```
1      seq
2         ... action 1 ...
3         ... action 2 ...
4         ...
5         ... action n ...
6      endseq
```

The `seq` block itself has type `Stmt`. The items in a block can have type `Action` or the type `Stmt`, *i.e.,* `seq-endseq` blocks can be nested.

The testbench in Section 4.7 contains an example of a `seq` block.

## 9.5 `StmtFSM`: conditionals (if-then-else)

Conditional process execution can be expressed with traditional if-then-else notation:

```
1   if ... Bool expression ...
2      ... expression of type Stmt ...
3   else
4      ... expression of type Stmt ...
```

As usual, if-then-elses can be be nested.

NOTE:

In Section 4.11 we described ordinary BSV if-then-else expressions, which often represent hardware multiplexers, where both arms are "evaluated" (the hardware exists for both sides) and the multiplexer merely selects one of the two outputs.

`StmtFSM` uses the same notation, but here it represents a *process*, and only one of the two arms is executed (like if-then-else in most software programming languages).

There is no ambiguity in these two uses of if-then-else notation—the context always clearly distinguishes what we mean, because there is no overlap between ordinary expressions and `StmtFSM` constructions.

## 9.6 `StmtFSM`: while-loops

Repetitive processes can be expressed with traditional while-loop notation:

```
1  while (... Bool expression ...)
2     ... expression of type Stmt ...
```

## 9.7 `StmtFSM`: pausing until some condition holds

An action in a `StmtFSM` can be the `await(b)` action, which simply waits until the boolean expression in its argument evaluates to true:

```
1  await (... Bool expression ...);
```

Of course, the `StmtFSM` itself cannot cause the value the change, since it is paused, and cannot change any state that would cause the expression to change its value. The state-change thus has to be effected by some other part of the BSV design (a concurrent FSM), not this particular `StmtFSM`.

## 9.8 `StmtFSM`: `mkAutoFSM`: a simple FSM module constructor

Creating an FSM using `StmtFSM` in BSV is a two-step process:

- Define the desired FSM behavior as an entity of type `Stmt`. Think of this as a *specification* of desired behavior.
- Instantiate a module that takes this specification and implements the behavior.

There are several `Stmt` $\longrightarrow$ module constructors available in the BSV library. In this book we use only one of them, `mkAutoFSM`:

```
1  mkAutoFSM (... argument expression of type Stmt ...);
```

This creates an FSM with the behavior specified by the `Stmt` argument. The FSM starts running immediately as we come out of reset, starting at the first statement, and terminates when we fall through the last statement. Of course, it may never terminate if it contains an infinite `while` loop.

Note: "terminating" the FSM in simulation means it executes a `$finish()` action which stops the complete simulation. In hardware (where there is no concept of `$finish()`) the FSM simply goes idle.[2] There may be other FSMs and rules in the design that continue running.

## 9.9 `StmtFSM`: many more features

Here we have covered all the features of `StmtFSM` that we need for coding Drum. There are many additional features that we skip here, such as for-loops and repeat-loops, fork-join parallel blocks, different kinds of modules instantiating `Stmt`s, predicated modules, and more. We refer the reader to the FSM chapter in the *bsc Libraries Reference Guide* [14].

The type `Stmt` is a first-class type in BSV. You can write functions that have arguments and results of type `Stmt`. For example, you might write a function that takes a numeric argument and returns a `Stmt`; this function can be invoked more than once to produce multiple `Stmt`s that differ because of the parameter. Each of these `Stmt`s can in instantiated into its own FSM. This further emphasizes the view that `Stmt` is a specification of an FSM behavior that is then instantiated in a module to produce that behavior.

[2] Technically, none of the rules implementing the FSM can fire any more.

# Chapter 10

# RISC-V: the Drum unpipelined CPU (an FSM)

## 10.1 Introduction

In this chapter we use code Drum's behavior, illustrated in Figure 10.1, using BSV's `StmtFSM` construct.

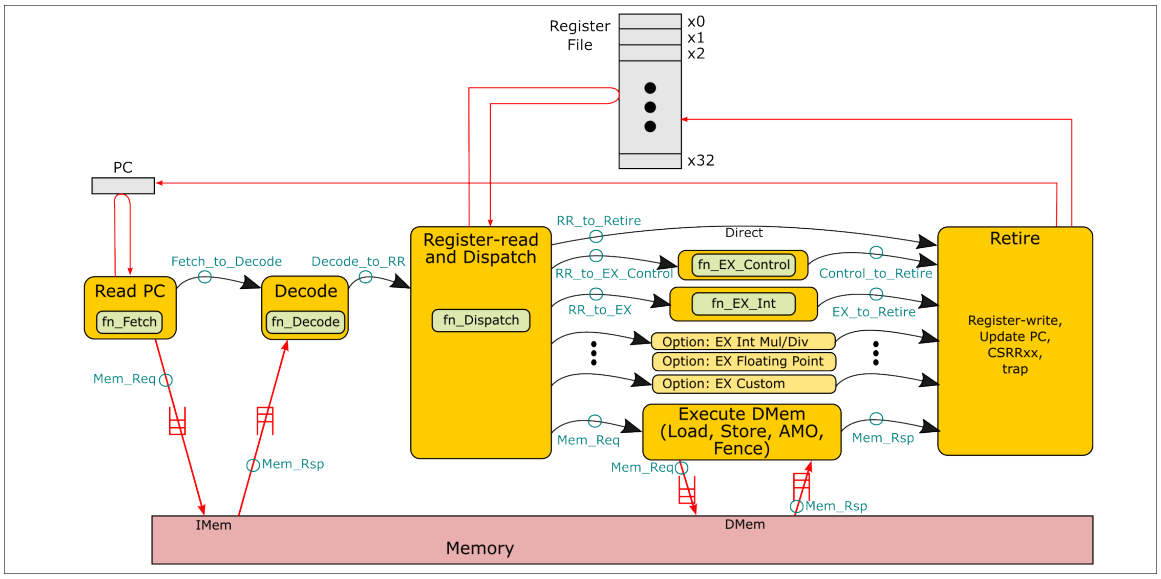



Figure 10.1: Simple interpretation of RISC-V instructions (same as Fig. **??**)

## 10.2 The Drum CPU module interface

The Drum CPU interface is shown below.

src_Common/CPU_IFC.bsv: line 27 ...

```
1 interface CPU_IFC;
2    method Action init (Initial_Params initial_params);
```

```
3
4     interface FIFOF_O #(Mem_Req) fo_IMem_req;
5     interface FIFOF_I #(Mem_Rsp) fi_IMem_rsp;
6     ...
7     interface FIFOF_O #(Mem_Req) fo_DMem_req;
8     interface FIFOF_I #(Mem_Rsp) fi_DMem_rsp;
9  endinterface
```

The interface is simple:

- The `init` method carries an `Initial_Params` struct containing any initial values needed by the CPU. A typical field is the initial value of the PC, since different software systems make different assumptions about the "starting address" for code. In many RV32I example codes, the starting address is `'h_8000_0000`.
- A `FIFOF_O` interface to carry memory requests for instructions (out-bound from the CPU to the memory);
- A `FIFOF_I` interface to carry corresponding memory responses containing instructions (in-bound from memory to the CPU);
- A `FIFO_O` interface to carry memory requests from load/store instructions (out-bound from the CPU to the memory);
- A `FIFOF_I` interface to carry corresponding load/store memory responses (in-bound from memory to the CPU).

(Please re-read Section 5.3.1 for the discussion on Harvard architectures, which have separate memory-access channels for instructions (Fetch, IMem) and for data (LOAD/STORE, DMem)).

Later we will see that Fife shares this interface, so that the Fife and Drum CPUs are easily interchangeable in any system or testbench.

## 10.3 The Drum CPU module

The STATE and INTERFACE sections of the Drum CPU module are shown below (we will discuss the elided BEHAVIOR section shortly).

src_Drum/CPU_FSM.bsv: line 50 ...

```
1  (* synthesize *)
2  module mkCPU (CPU_IFC);
3     // ================================================================
4     // STATE
5
6     // Don't run until the PC (and other things) are initialized
7     Reg #(Bool) rg_running <- mkReg (False);
8     ...
9     // The Program Counter
10    Reg #(Bit #(XLEN)) rg_pc   <- mkReg (0);
11
12    // The integer register file
13    RISCV_GPRs_IFC #(XLEN)  gprs <- mkRISCV_GPRs_synth;
14
15    // CSRs
```

```
16    CSRs_IFC csrs <- mkCSRs;
17
18    // Inter-step registers
19    Reg #(Fetch_to_Decode)      rg_Fetch_to_Decode      <- mkRegU;
20    Reg #(Decode_to_RR)         rg_Decode_to_RR         <- mkRegU;
21    Reg #(Result_Dispatch)      rg_Dispatch             <- mkRegU;
22    Reg #(EX_Control_to_Retire) rg_EX_Control_to_Retire <- mkRegU;
23    Reg #(EX_to_Retire)         rg_EX_to_Retire         <- mkRegU;
24
25    // Paths to and from memory
26    FIFOF #(Mem_Req) f_IMem_req  <- mkFIFOF;
27    FIFOF #(Mem_Rsp) f_IMem_rsp  <- mkFIFOF;
28
29    FIFOF #(Mem_Req) f_DMem_req  <- mkFIFOF;
30    FIFOF #(Mem_Rsp) f_DMem_rsp  <- mkFIFOF;
31
32    // Regs to set up exception handling
33    Reg #(Bool)        rg_exception <- mkReg (False);
34    Reg #(Bit #(XLEN)) rg_epc       <- mkRegU;
35    Reg #(Bit #(4))    rg_cause     <- mkRegU;
36    Reg #(Bit #(XLEN)) rg_tval      <- mkRegU;
37
38    // ****************************************************************
```

*(... BEHAVIOR elided, to be discussed shortly ...)*

src_Drum/CPU_FSM.bsv: line 357 ...

```
1     // ****************************************************************
2     // INTERFACE
3
4     method Action init (Initial_Params initial_params);
5        ...
6        rg_pc      <= initial_params.pc_reset_value;
7        rg_running <= True;
8     endmethod
9
10    interface fo_IMem_req = to_FIFOF_O (f_IMem_req);
11    interface fi_IMem_rsp = to_FIFOF_I (f_IMem_rsp);
12    ...
13    interface fo_DMem_req = to_FIFOF_O (f_DMem_req);
14    interface fi_DMem_rsp = to_FIFOF_I (f_DMem_rsp);
15 endmodule
```

The STATE section first instantiates a register `rg_running`, initially False, that will be set to True by the `init` method that initializes the PC to a specific initial value. The FSM will wait for this before starting the first instruction-fetch.

Then, `mkCPU` instantiates a register for the PC, and then the GPRs module (desribed in Section 8.2) and the CSRs modules (described in Section 8.3). Then, it instantiates a set of registers to hold values between temporal FSM steps (with struct types shown in Figure 10.1). For example, the Fetch step will write a value into `rg_Fetch_to_Decode` which will be read later by the Decode step. Then, it instantiates four FIFOFs for IMem requests

(outgoing) and responses (incoming), and for DMem requests (outgoing) and responses (incoming). As mentioned before, we do not make any assumption about the *latency* of memory requests, *i.e.,* how long it takes the external memory subsystem to consume a request from one of the request FIFOs and enqueue a response into the corresponding response FIFO. Finally, it instantiates the four registers needed for trap-handling, described in Section 2.7.

In the display above we have elided the BEHAVIOR section of the module, which we will describe shortly.

In the INTERFACE section, the `init` method initializes the PC and sets `rg_running` to true, releasing the BEHAVIOR section to start executing. We use the interface transformers discussed in Section 7.5.5 that produce Semi-FIFO "views" of FIFOs to lift the FIFO interfaces into the module interface.

## 10.4 Help-functions for the Drum CPU module behavior

Before we look at the FSM implementing Drum behavior, we first have some `Action` functions that encapsulate some common actions performed in several states in the FSM.

NOTE: In BSV, function definitions do not have to be at the top-level of a file, in fact they can be defined at any nested level. Here, we define these inside a module.

The following function writes an rd-value into the the GPRs, in those cases where the instruction has an rd (`fn_Decode`, described in Section 6.3, computed `has_rd` for each kind of instruction):

src_Drum/CPU_FSM.bsv: line 95 ...

```
function Action fa_update_rd (RR_to_Retire x1,
                              Bit #(XLEN) rd_val);
   action
      if (x1.has_rd) begin
         let rd = instr_rd (x1.instr);
         gprs.write_rd (rd, rd_val);
         ...

      end
   endaction
endfunction
```

The following function saves values into registers `rg_epc`, `rg_cause` and `rg_tval`, when a trap is detected. These will later be written into the corresponding CSR registers during trap-handling.

src_Drum/CPU_FSM.bsv: line 111 ...

```
function Action fa_setup_exception (Bit #(XLEN) epc,
                                    Bit #(4) cause,
                                    Bit #(XLEN) tval);
   action
      rg_exception <= True;
      rg_epc       <= epc;
```

```
7           rg_cause     <= cause;
8           rg_tval      <= tval;
9        endaction
10    endfunction
```

This function is the last action during an instruction's execution, updating the PC and the instruction number:

src_Drum/CPU_FSM.bsv: line 122 ...

```
1     function Action fa_redirect_Fetch (Bit #(XLEN) next_pc);
2        action
3           rg_pc   <= next_pc;
4           rg_inum <= rg_inum + 1;
5        endaction
6     endfunction
```

## 10.5 The Drum CPU module behavior

The Drum CPU module BEHAVIOR is defined in two parts. First we define an FSM specification `exec_one_instr` (of type `Stmt`) for executing a single instruction. Then, we embed that `Stmt` in a next-level `Stmt` in an infinite while-loop, preceded by an `await` action that waits for the initial PC to be set by the `init` method. Finally, we instantiate that `Stmt` using a `mkAutoFSM` module that will implement the behavior starting immediately when emerging from reset. The FSM never stops, because of the infinite while-loop.

```
module mkCPU (CPU_IFC);
   // STATE
   ...
   // ****************************************************************
   // BEHAVIOR
   ...
```

src_Drum/CPU_FSM.bsv: line 129 ...

```
1     // ================================================================
2     // FSM for Drum behavior
3
4     Stmt exec_one_instr =
5     ...
```

src_Drum/CPU_FSM.bsv: line 352 ...

```
1     mkAutoFSM (seq
2                   await (rg_running);
3                   while (True) exec_one_instr;
4                endseq);
```

```
   // ****************************************************************
   // INTERFACE
   ...
endmodule
```

---

**Exercise 10.1:**

What might happen if we omitted the "`await!(rg_running)`" statement in the Drum CPU? (Try it in simulation!)

*Hint:* The FSM may start running before the PC has been initialized ...

□

---

### 10.5.1 FSM action for Fetch

The first action inside `exec_one_instr` is for Fetch:

src_Drum/CPU_FSM.bsv: line 137 ...

```
1   // ================================================================
2   // Fetch
3   action
4      ...
5      let y <- fn_Fetch (rg_pc,
6          ...
7      rg_Fetch_to_Decode <= y.to_D;
8      f_IMem_req.enq (y.mem_req);
9      ...
10
11  endaction
```

This applies `fn_Fetch()` (Section 6.2) to the PC, stores the `to_D` part of the result in register `rg_Fetch_to_Decode`, and sends the `mem_req` part of the result to the Instruction Memory by enqueueing it on the outgoing `f_IMem_req` FIFO. The other end (dequeue end) of the FIFO is in the module interface. The outer environment of this module will dequeue it and forward it to memory.

### 10.5.2 FSM action for Decode

The next action in the FSM is the Decode step:

src_Drum/CPU_FSM.bsv: line 154 ...

```
1   // ================================================================
2   // Decode
3   action
4      let mem_rsp <- pop_o (to_FIFOF_O (f_IMem_rsp));
5      let y       <- fn_Decode (rg_Fetch_to_Decode, mem_rsp, rg_flog);
6      rg_Decode_to_RR <= y;
7      ...
8
9   endaction
```

We pop the response from IMem (bind `mem_rsp` to the value at head of the FIFO, and also remove it from the FIFO), then apply `fn_Decode()` (Section 6.3) to the `Fetch_to_Decode`

value from the Fetch step and the IMem response, and store the function result in register `rg_Decode_to_RR`.

Recall that each `Action` is instantaneous, and the FSMs sequence actions. In the code so far, the Fetch action takes place in one instant, and the Decode action takes place at a later instant. The Decode action is not "enabled" until an IMem response is available in the `f_IMem_Rsp` FIFO; it will simply wait until a response is available. Thus, the time interval between the Fetch instant and the Decode instant is unpredictable; it depends on when the IMem result becomes available (see dicussion in Section 3.2.1 on unpredictable memory latency).

NOTE:

Looking ahead to a topic we'll discuss in more detail in Chapter 14, each module interface method has a so-called *implicit condition*, *i.e.,* an accompanying boolean value value that indicates when the method is "ready" or not or, to say it another way, whether the method is "enabled" or not. For the FIFO methods "`.first`" and "`.deq`", which are used by "`pop_o`" above, the methods are ready/enabled only when the FIFO is not empty.

The Decode action in the FSM is translated by the *bsc* compiler into a BSV *rule*. A rule does not "fire" until all the implicit conditions in its method-calls are enabled. This is why the Decode action implicitly waits until something is available in the FIFO.

### 10.5.3 FSM action for Dispatch

The next FSM action is the Register-Read and Dispatch step:

src_Drum/CPU_FSM.bsv: line 166 ...

```
1     // ================================================================
2     // Register-Read and Dispatch
3     action
4        // Read GPRs
5        // Ok that read_rs1 and read_rs2 may return junk values
6        //         since not all instrs have rs1/rs2.
7        let x       = rg_Decode_to_RR;
8        let rs1_val = gprs.read_rs1 (instr_rs1 (x.instr));
9        let rs2_val = gprs.read_rs2 (instr_rs2 (x.instr));
10
11       Result_Dispatch y <- fn_Dispatch (x, rs1_val, rs2_val, rg_flog);
12       rg_Dispatch       <= y;
13       ...
14
15    endaction
```

Using the information in `rg_Decode_to_RR` created in the Decode step, we read the two registers `rs1` and `rs2`. We apply the function `fn_Dispatch` (Section 6.4) and store the result in register `rg_Dispatch`.

Note that we blindly read both registers `rs1` and `rs2`, even though many instructions do not have one or both of these fields. In those situations we'll be reading bogus/irrelevant values, but it does not matter—in the Execute step we will only use these values in instructions that need them. Recall that while in a software programming language this may seem

unnecessary or wasted work, in this hardware context nothing is wasted—the hardware for reading both registers exist, there is nothing lost in using it.

### 10.5.4 FSM actions for Execute and Retire

After the Dispatch FSM step, we move on to the Exectute and Retire FSM steps. Figure 10.2 shows the details of the four "flows" that may follow. Each of the flows can have an exception

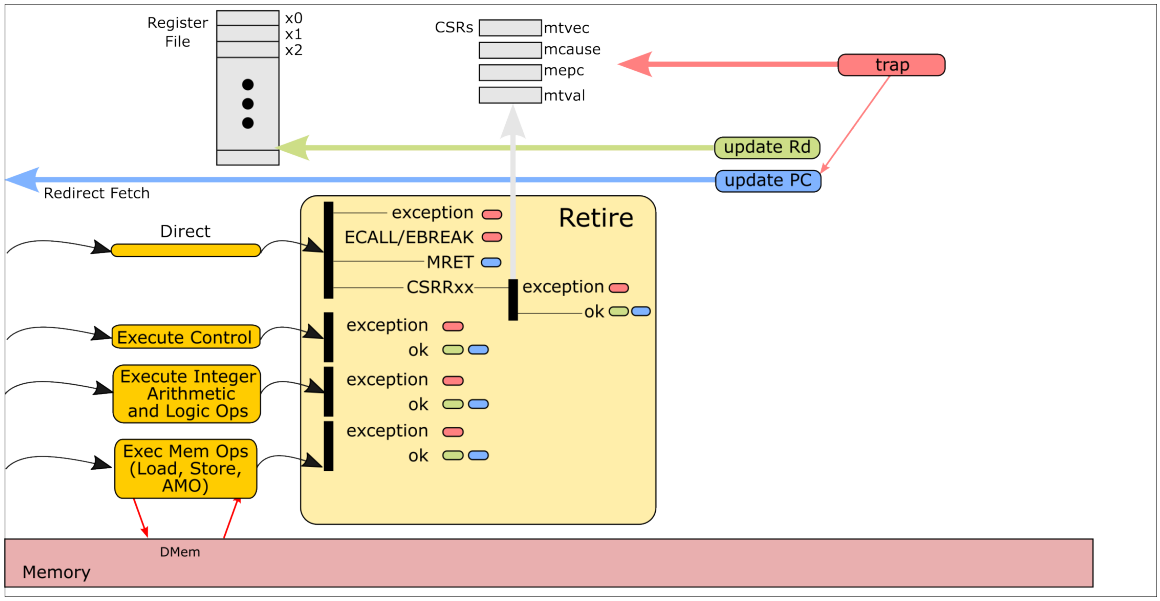



Figure 10.2: Execute and Retire actions in Drum

or a normal (OK) result. The Direct flow may need to execute a SYSTEM instruction (ECALL, EBREAK, MRET and CSRRxx). Executing a CSRRxx instruction may also have an exception or normal result. For each flow, the colored ovals show the actions may be performed; these correspond exactly to the three help-functions discussed earlier in Section 10.4.

The FSM code follows the structure shown in the diagram, and have the following broad structure:

src_Drum/CPU_FSM.bsv: line 192 ...

```
1       if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_DIRECT)
2           ...
3       else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_CONTROL)
4           ...
5       else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_INT)
6           ...
7       else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_DMEM)
```

We next elaborate each of the four flows (the elided arms in the above if-then-else).

**FSM actions in Direct flow of Execute and Retire**

src_Drum/CPU_FSM.bsv: line 192 ...

```
      if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_DIRECT)
         action
            let x_direct = rg_Dispatch.to_Retire;
            if (x_direct.exception)
               fa_setup_exception (x_direct.pc,        // epc
                                   x_direct.cause,     // cause
                                   x_direct.tval);     // tval
            else if (is_legal_CSRRxx (x_direct.instr)) begin
               match { .exc, .rd_val } <- csrs.mav_csrrxx (x_direct.instr,
                                                           x_direct.rs1_val);
               if (exc)
                  fa_setup_exception (x_direct.pc,                  // epc
                                      cause_ILLEGAL_INSTRUCTION,    // cause
                                      x_direct.instr);              // tval
               else begin
                  fa_update_rd (x_direct, rd_val);
                  fa_redirect_Fetch (x_direct.fallthru_pc);
               end
            end
            else if (is_legal_MRET (x_direct.instr))
               fa_redirect_Fetch (csrs.read_epc);
            else if (is_legal_ECALL (x_direct.instr)
                     || is_legal_EBREAK (x_direct.instr)) begin
               let cause = ((x_direct.instr [20] == 0)
                            ? cause_ECALL_FROM_M
                            : cause_BREAKPOINT);
               fa_setup_exception (x_direct.pc,    // epc
                                   cause,
                                   0);             // tval
            end
            else begin
               wr_log (rg_flog, $format ("CPU.EX.Direct: IMPOSSIBLE"));
               $finish (1);
            end
```

First, for convenience, we give the value `rg_Dispatch.to_Retire` a shorter and more convenient name, `x_direct`, to be used in the rest of this `action-endaction` block.

If we received an exception from the Dispatch step, we invoke the help-function `fa_setup_exception()` (Section 10.4) to set register `rg_exception` to true and to record the relevant values in the registers `rg_epc`, `rg_epc`, `rg_cause` and `rg_tval`.

If we received a CSRRxx instruction from the Dispatch step, we invoke the `mav_csrrxx()` method in the `csrs` module to perform the instruction on the relevant CSRs. The result is a 2-tuple (Section **??**) and we use a `match` construct bind the names `exc` and `rd_val` to the two components. If the CSRRxx instruction failed (`exc` is true, once again we invoke `fa_setup_exception()` to record this. Otherwise, we invoke `fa_update_rd()` to write `rd_val` to a GPR, and we invoke `fa_redirect_Fetch()` to update the PC to the fall-through PC value.

If we received an MRET instruction from the Dispatch step, we simply invoke `fa_redirect_Fetch()` to resume executing at the PC value provided in `rg_mtvec`.

If we received an ECALL or EBREAK instruction from the Dispatch step, recall from Section 2.7.1 that these instructions merely invoke exceptions, with "cause codes" `cause_ECALL_FROM_M` and `cause_BREAKPOINT`, respectively (Section 2.7 discussed cause codes). We simply invoke `fa_setup_exception()` to record this. Inside this clause, since we are already assured that the current insruction is either ECALL or EBREAK, we only need to check one bit (`instr[20]`) to distinguish them in order to select the correct cause code.

There is a final "`else`" clause, but between the functionality of the Decode and Dispatch steps, it should be impossible to enter this clause (this is just a bit of defensive programming and can safely be removed; even if not removed, it does not produce any hardware).

**FSM actions in Execute Control flow of Execute and Retire**

src_Drum/CPU_FSM.bsv: line 228 ...

```
      else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_CONTROL)
         seq
            // ---------------- Execute
            action
               let y <- fn_EX_Control (rg_Dispatch.to_EX_Control, rg_flog);
               rg_EX_Control_to_Retire <= y;
            endaction
            // ---------------- Retire
            action
               let x_direct  = rg_Dispatch.to_Retire;
               let x_control = rg_EX_Control_to_Retire;
               if (x_control.exception)
                  fa_setup_exception (x_direct.pc,
                                      x_control.cause,
                                      x_control.tval);
               else begin
                  fa_update_rd (x_direct, x_control.data);
                  fa_redirect_Fetch (x_control.next_pc);
               end
               ...
            endaction
         endseq
```

This sub-FSM is itself an FSM with a sequence of two actions, one to execute and one to retire. The Execute action invokes `fn_EX_Control` (Section 6.5) on the information provided by Dispatch, and stores the result in register `rg_EX_Control_to_Retire`.

The Retire action checks if that resulted in an exception or not. If an exception, it invokes `fa_setup_exception` to record that. If not an exception, it invokes `fa_update_rd` to store the output value (this can only be a "return address" computed for JAL and JALR), and it invokes `fa_redirect_Fetch` to resume execution at the `next_pc` value computed by BRANCH/JAL/JALR.

**FSM actions in Execute Integer flow of Execute and Retire**

src_Drum/CPU_FSM.bsv: line 271 ...

```
      else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_INT)
         seq
```

```
            // ---------------- Execute
            action
               let y <- fn_EX_Int (rg_Dispatch.to_EX, rg_flog);
               rg_EX_to_Retire <= y;
               ...
            endaction
            // ---------------- Retire
            action
               if (rg_EX_to_Retire.exception)
                  fa_setup_exception (rg_Dispatch.to_Retire.pc,
                                      rg_EX_to_Retire.cause,
                                      rg_EX_to_Retire.tval);
               else begin
                  fa_update_rd (rg_Dispatch.to_Retire, rg_EX_to_Retire.data);
                  fa_redirect_Fetch (rg_Dispatch.to_Retire.fallthru_pc);
               end
            endaction
         endseq
```

This sub-FSM is similar to the Control sub-FSM: it is itself an FSM with a sequence of two actions, one to execute and one to retire. The Execute action invokes `fn_EX_Int` (Section 6.6) on the information provided by Dispatch, and stores the result in register `rg_EX_to_Retire`.

The Retire action checks if that resulted in an exception or not. If an exception, it invokes `fa_setup_exception` to record that. If not an exception, it invokes `fa_update_rd` to store the output value and `fa_redirect_Fetch` to resume execution at the fall-through PC.

**FSM actions in DMem flow of Execute and Retire**

src_Drum/CPU_FSM.bsv: line 293 ...

```
      else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_DMEM)
         seq
            // ---------------- Execute: send DMem req
            action
               Mem_Req y = rg_Dispatch.to_EX_DMem;
               f_DMem_req.enq (y);
               ...
            endaction
            // ---------------- Retire: process DMem rsp
            action
               let x_direct = rg_Dispatch.to_Retire;
               let mem_rsp <- pop_o (to_FIFOF_O (f_DMem_rsp));
               Bool exception = ((mem_rsp.rsp_type == MEM_RSP_ERR)
                                 || (mem_rsp.rsp_type == MEM_RSP_MISALIGNED));
               if (exception) begin
                  Bit #(4) cause = ((mem_rsp.rsp_type == MEM_RSP_MISALIGNED)
                                    ? (is_LOAD (x_direct.instr)
                                       ? cause_LOAD_ADDRESS_MISALIGNED
                                       : cause_STORE_AMO_ADDRESS_MISALIGNED)
                                    : (is_LOAD (x_direct.instr)
                                       ? cause_LOAD_ACCESS_FAULT
                                       : cause_STORE_AMO_ACCESS_FAULT));
```

```
23                fa_setup_exception (x_direct.pc,                 // epc
24                                    cause,
25                                    truncate (mem_rsp.addr));    // tval
26             end
27             else begin
28                fa_update_rd (x_direct, truncate (mem_rsp.data));
29                fa_redirect_Fetch (x_direct.fallthru_pc);
30             end
31             ...
32          endaction
33       endseq
```

This sub-FSM is similar to Control and Integers sub-FSMs: it is itself an FSM with a sequence of two actions. The Execute action just sends the memory-request provided by the Dispatch stage to memory by enqueueing it on the outgoing `f_DMem_req` FIFO.

The Retire action pops the mememory response from the incoming `f_DMem_rsp` FIFO. If the memory returned an exception, we compute the proper cause code and then invoke `fa_setup_exception()` to record it. If the memory did not return an exception, we invoke `fa_update_rd()` to store any loaded value into the GPRs, and invoke `fa_redirect_Fetch()` to resume execution at the fall-through PC.

### 10.5.5 FSM actions for exceptions

src_Drum/CPU_FSM.bsv: line 340 ...

```
1     if (rg_exception)
2        action
3           Bool is_interrupt = False;
4           Bit #(XLEN) tvec_pc <- csrs.mav_exception (rg_epc,
5                                                      is_interrupt,
6                                                      rg_cause,
7                                                      rg_tval);
8           rg_exception <= False;
9           fa_redirect_Fetch (tvec_pc);
10       endaction
```

The final action in the `exec_one_instr` FSM checks register `rg_exception` to see if any of the Retire flows recorded an exception. If so, it invokes the `mav_exception()` method in the CSRs module, and invokes `fa_redirect_Fetch` so that execution resumes at the trap-vector PC provided in CSR `mt_vec`.

## 10.6 Conclusion

And that is the complete Drum CPU. We can compile it, connect the CPU interface to a memory system, and run the system on RISC-V programs compiled for the RV32I ISA subset (and, further, Drum can recover from illegal instructions due to trap-handling).

### 10.6.1 But Drum code looks just like C!? Why not code it in C?

Looking at the BSV FSM code for Drum, it seems like we could convert it into C code by simply replacing a few BSV idioms with corresponding C idioms (like replacing `begin-end` and `action-endaction` with "`{`" and "`}`", replacing BSV register declarations with C variable declarations, *etc.*). Similarly, each of the pure functions `fn_Fetch`, `fn_Decode`, ... can easily be slightly tweaked into C functions.

This can indeed be done, and the result would be a C simulator for RISC-V! We could compile it with any C compiler and run the software on any platform.

So, why not code Drum in C instead of BSV, and change the *bsc* compiler to accept C syntax? It is difficult to compile general-purpose C into hardware. C has many constructs that have no obvious mapping into hardware, such as pointers, the address-of operator "`&`", address arithmetic, `malloc`, and so on. But can't we define a restricted subset of C suitable for hardware design? Compilation is still very difficult. In the suggested re-coding in C above, we lose the distinction between variables used to hold state (registers, register files, FIFOs) and ordinary variables for temporary values (which in hardware are just wires). We also lose the distinction between non-temporal statements (statements within an action block) versus temporally sequenced (actions in an FSM). A C-to-hardware compiler would have to reconstruct this information.

But the killer reason is sequentiality *vs.* concurrency. With Drum it is deceptively easy to imagine a simple correspondence with C, because Drum is a sequential FSM, and C is a sequential language. When we advance to concurrent FSMs, including pipelined processors like Fife, we depart subtantially and deeply from sequential process semantics, and C becomes more and more unsuitable for hardware description.

NOTE: There are products in the marketplace under the rubric of "High Level Synthesis" (HLS) that translate C source codes into synthesizable Verilog. These indeed work on a carefully defined subset of C, not the full C language. They work best on simple programs that contain nested loops working on dense, rectangular arrays (many image-processing and linear algebra applications can be so characterized), because the compiler is able to analyze the originally sequential semantics of such programs and transform them into highly parallel, highly structured representations that can then be mapped into very stylized hardware (data path plus control FSM). Even with this capability, the programmer may need to provide many additional directives to the compiler to guide it towards good quality hardware. We do not consider this to be a general purpose approach to hardware design.

# Chapter 11

# BSV: Packages, Files and Imports; System Top-Level


## 11.1 Packages and Files

## 11.2 Importing C; Fife/Drum memory system

## 11.3 `Top.bsv` instantiating `CPU.bsv` and `Mems_Devices.bsv`

# Chapter 12

# RISC-V: the Fife pipelined CPU: Principles

## 12.1 Introduction

In this chapter we turn our attention to Fife, the pipelined CPU. Our focus here is purely on identifying new problems raised by pipelining, and proposing solutions. In the next chapter we will look at BSV code to implement Fife.

Figure 12.1 annotates the abstract execution algorithm in Figure 3.1 with some specifics for the pipelined implementation in Fife. Unlike Drum, where each yellow box was one step in

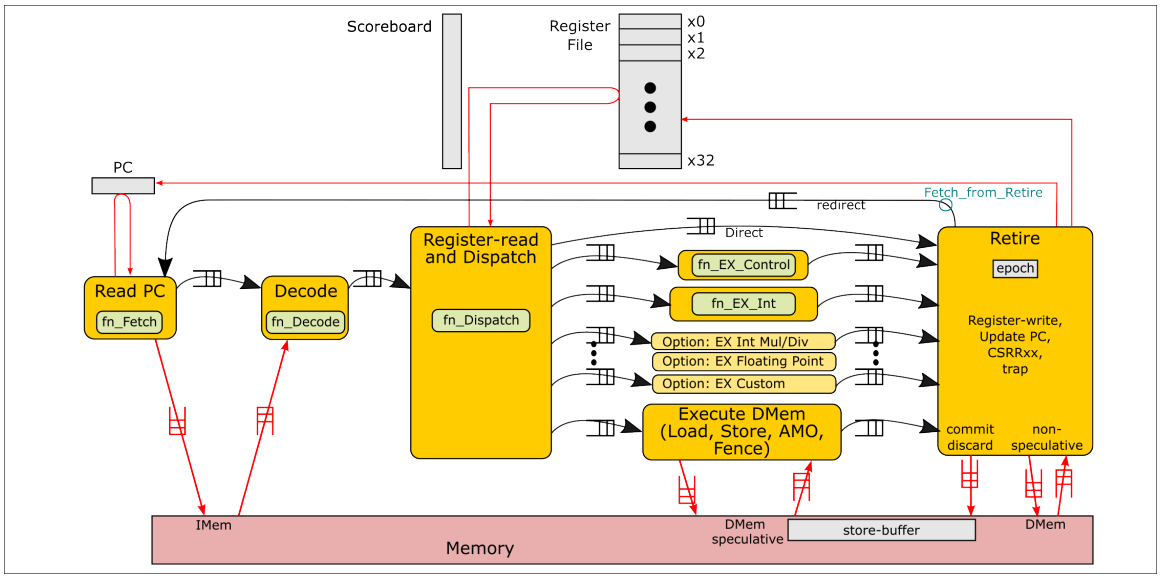



Figure 12.1: Pipelined interpretation of RISC-V instructions (Fig. 3.1 with some annotations)

a sequential process, now we interpret each yellow box as containing its own infinite process. There are now half-a-dozen or more processes in the diagram (one for each yellow box), all running *concurrently*.

For each of the yellow boxes, we use the word "*step*" for Drum and "*stage*" for Fife. For Fife, each black arrow in the diagram represents a flow of *messages* sent from one stage to another. These messages are sent via FIFO buffers (depicted by ⊐⊏ annotations in the figure).

Each stage is an infinite loop, consuming incoming messages and producing outgoing messages. Thus, while the Retire stage is working on instruction $n$, the Execute, Register-Read, Decode and Fetch stage(s) may be working on instructions $n+1$, $n+2$, $n+3$, and $n+4$, respectively. Thus, there is a sequence, or train, of instructions flowing through the diagram from left to right.

Pipelining raises four new problems, and these are the focus of this chapter:

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

## 12.2 Keeping the Fetch Stage Working with PC Prediction and Epochs

What should the Fetch stage do after issuing a request to IMem for instruction $n$? To issue another request, it needs to know the PC of instruction $n+1$, but there are several uncertainties about that next PC:

- The current Fetch itself can raise an exception (trap) if its PC is misaligned, is an unsupported memory address, *etc.*. In this case the next PC will the trap-handler PC instead of the "normal" next PC.
- If it is a BRANCH instruction, until it reaches the Execute Control stage we do not know the branch target PC address, nor if the branch is taken or not.
- If it is a JAL or JALR instruction, until it reaches the Execute Control stage we do not know the target PC address.
- Many instructions can raise an exception (trap) (illegal instructions, BRANCH/JAL/JALR with misaligned target addresses, DMem ops with misaligned addresses or unsupported addresses, *etc.*); in these cases the next PC will the trap-handler PC instead of the "normal" next PC.
- The CPU may choose to respond to an external interrupt, in which case the next PC will be the interrupt-handler's address.

Note, the Fetch stage knows nothing about instruction $n$ other than its PC. The instruction itself is not known until IMem sends its response to the Decode stage (and assuming the Fetch does not raise an exception).

### 12.2.1 PC Prediction in the Fetch Stage

A standard solution is for the Fetch stage to *predict* the next PC, *i.e.,* make a guess about the next PC. Since all RISC-V RV32 instructions are 32-bits wide (4 bytes), and *most* of

them "fall-through" to the next adjacent instruction, a simple prediction is: PC+4. This prediction will be correct for most instructions, but will be wrong for BRANCH instructions that take the branch, for JAL/JALR instructions, and for any instruction that traps. When the prediction is wrong, the instructions that follow immediately are called "mispredicted" or "wrong-path" instructions.

NOTE:

RISC-V instructions are all 32-bits wide, so PC+4 is a reasonably good guess. In ISAs that have variable-length instructions, prediction may be more complicated. Even in RISC-V, when implementing the "C" (Compressed Instructions) extension, some instructions may be 16-bits wide, raising similar complications.

Earlier we said "the Fetch stage knows nothing about instruction $n$ other than its PC". This is not strictly true–the CPU may have fetched this PC before (*e.g.,* this PC is inside a loop, or in a procedure that is called repeatedly). Knowledge of past behavior can improve the current prediction. Most predictors in modern processors use past history to improve and "tune" their branch predictors dynamically while executing the program. Designing good branch predictors is a deep topic for which there are many good textbooks (for example, [6]).

PC prediction can be seen as a kind of "machine learning". The CPU's past execution history constitutes the "training data" for a model, and the model is then asked to predict the next PC for the current PC.

### 12.2.2 Identifying and Flushing Wrong-path Instructions

Clearly, we need to identify and flush wrong-path instructions from the pipeline.

Suppose the Fetch stage issues requests for two instructions $i_1$ at address $a_1$ and $i_2$ at address $a_2$, where $a_2$ is predicted from $a_1$. When issuing the request for $a_1$, the Fetch stage can pass along $a_2$ to the Decode stage, from which point it can accompany $i_1$ as it journeys through the pipeline (*i.e.,* every instruction is accompanied by its next-PC prediction).

When $i_1$ reaches the Retire stage, we know the *correct* next PC (Trap handler PC? PC+4? Branch-taken target? JAL/JALR target?). By comparing this actual next PC with $a_2$, we know whether the successor to $i_1$ was predicted correctly or not.

If we find that the prediction was correct, there is nothing more to be done; we allow the pipeline flow to proceed.

If we find that the prediction was wrong, then two things must happen:

- We need to *redirect* the Fetch stage to start fetching from the correct next-PC. This involves sending a message from the Retire stage back to the Fetch stage containing the correct next-PC. Suppose the first instruction fetched after this redirection is $j_1$.
- Instruction $i_2$, and possibly following instructions $i_3$, $i_4$, ... until $j_1$ are wrong-path instructions, and must be flushed.

When the Retire stage starts flushing wrong-path instructions $i_2$, $i_3$, $i_4$, ... how does it know when it has reached the end of the wrong-path sequence? In other words, how does it know when it sees $j_1$? This is precisely the purpose of the `rg_epoch` register shown in Figure 12.1.

Think of `rg_epoch` as a counter that continuously counts upward. Suppose the current value is $e_1$. As described above, when the Retire stage recognizes an instruction whose successor has been mispredicted, we send a redirection message to the Fetch stage with the corrected PC. The Retire stage also increments $e_1$ and sends the incremented value as part of the redirection. Each time the Fetch stage is redirected, it remembers the new epoch value. It also sends this value down the pipeline, accompanying each instruction fetched with this value.

Now, flushing wrong-path instructions in the Regire stage is easy: $i_2$, $i_3$, $i_4$, ... will be accompanied by the old epoch value $e_1$, whereas the first correct-path instruction $j_1$ will be accompanied by the new epoch value $e_1 + 1$. Thus, the Retire stage knows exactly which instructions are wrong-path and it can discard them.

---

**Exercise 12.1:**

We have describe `rg_epoch` as a counter that is incremented on each recognition of a misprediction. If the register contents have type `Bit #(n)`, then this will wrap-around to 0 after $2^n$ increments. Is this a problem?

**Exercise 12.2:**

If `rg_epoch` contains a `Bit #(n)` value, how small can $n$ be?

□

---

### 12.2.3 Terminology: Speculative Instructions and Commits

Before an instruction has reached the Retire stage, there is always the possibility that an earlier instruction that is ahead of it in the pipeline will branch/jump to a PC that was not predicted, or will trap, making this instruction irrelevant. Until this moment, we say that the instruction is still "*speculative*". When it reaches the Retire stage, we say that it's side-effects can be "*committed*".

### 12.2.4 Speculative instructions should not have any side-effects

It is not enough for the Retire stage just to discard mispredicted instructions. Instructions have side-effects: they may modify registers and write to memory. We must ensure that speculative instructions make no modifications that are visible to right-path instructions that follow, until they reach the Retire stage. The details of how this is accomplished will be seen in Section 12.5 and Section 12.6.

## 12.3 Managing Register Read/Write Hazards with a Scoreboard

Suppose instruction $i_1$ writes to register $x_7$, and the following instruction $i_2$ reads from register $x_7$. Instruction $i_1$'s write to $x_7$ only happens in the Retire stage. If $i_2$ were to follow behind $i_1$ immediately, it will be in the Exec stage, and would have already read $x_7$

earlier when it was in the Register-Read stage. In other words, it would have read a *stale* or obsolete value $x_7$. This is called a Read-Write *hazard*, or a read-after-write *dependency*.

In this situation, we need to make $i_2$ wait in the Register-Read stage until $i_1$ has completed its update of $x_7$. This is precisely the purpose of the `scoreboard` shown in Figure 12.1.

The `scoreboard` is an array of 32 1-bit registers (one bit for each GPR). When an instruction (such as $i_1$) passes through the Register-Read stage, if it writes to register $x_7$, we set the corresponding bit 7 in the scoreboard to 1, indicating that $x_7$ is "busy". When $i_1$ reaches the Retire stage and writes to the register, it also resets the scoreboard bit 7 to 0, indicating that $x_7$ is "not busy".

When an instruction (such as $i_2$) reaches the Register-Read stage and wants to read a register (such as $x_7$), if the corresponding scoreboard bit says it is busy, then the Register-Read stage "*stalls*", *i.e.,* it waits until the scoreboard condition is cleared (by $i_1$ in the Retire stage).

While $i_2$ is waiting in the stalled Register-Read stage, note that the following Execute stage may become "empty", *i.e.,* there is no instruction occupying that stage. We refer to this as a "*pipeline bubble*".

---

**Exercise 12.3:**

For two consecutive instructions $i_1$ and $i_2$,

- $i_1$ may want to write register $x_7$ and $i_2$ may want to read $x_7$,
- $i_1$ may want to write register $x_7$ and $i_2$ may write to write $x_7$,
- $i_1$ may want to read register $x_7$ and $i_2$ may want to read $x_7$,
- $i_1$ may want to read register $x_7$ and $i_2$ may want to write $x_7$.

Above, we motivated scoreboards with the first scenario. What about the other three scenarios?

**Exercise 12.4:**

Write-write hazards can be treated just like read-after-write hazards. Alternatively the 1-bit in the scoreboard for a register (say $x_7$) can be generalized into an $n$ bit up/downcounter, indicating the number of instructions that have been allowed into Execute pipelines that intend to write $x_7$. The Retire stage decrements this counter; the Register-Read stage stalls an instruction if these counters (for its input registers) are non-zero; and the Register-Read stage increments this counter for an instruction's destination register.

Implement a scoreboard module with this scheme. What should happen if a counter reaches its maximum value? How many bits should each counter have?

**Exercise 12.5:**

In the counter-based scoreboard of the previous exercise, if there are multiple instructions in the Execute stages that intend to write $x_7$, in what order can those writes occur? What would be the consequences of a wrong order?

*Hint:* The answer is in Section 12.4.

□

---

### 12.3.1 Releasing Scoreboard Reservations for Uncommitted Instructions

When the Retire stage discards an instruction due to mis-speculation, that instruction may be one which normally would have written a result value into register $x_j$ in the register file. If so, when it passed through the Register-Read-and-Dispatch stage, it would have marked register $x_j$ as "busy" in the scoreboard. Now, when we discard the instruction, even though we do not write any value into register $x_j$, we still need to release the "busy" reservation on $x_j$ (otherwise, any succeeding instruction trying to read $x_j$ will get stuck in the Register-Read stage. Said another way, the reservation in the scoreboard is a side-effect of this instruction that needs to be rolled back.

### 12.3.2 Bypassing

Digital hardware usually runs in time units of "clock cycles". The Retire stage writes a GPR (possibly) and writes the scoreboard (to mark it "not-busy"). The Register-Read stage reads zero to two GPRs, reads the scoreboard (to check "not-busy") and writes the scoreboard (to set "busy").

For ordinary registers a write is only visible on the next clock cycle. Thus, if the scoreboard is just an ordinary register, the Register-Read stage cannot observe "not-busy" until one clock after the Retire stage has marked "not-busy". This does not affect correctness, but slows the performance of the CPU.

It is possible to design some extra circuitry around the scoreboard so that the Register-Read stage can observe "not-busy" on the *same* clock cycle as when Retire marks it "not-busy". This technique is generically called "*bypassing*" or "*short-circuiting*".

---

**Exercise 12.6:**

Implement a scoreboard unit with bypassing/short-circuiting as described in the above note.

*Hint:* Needs BSV "Concurrent Registers" (advanced topic!)

**Exercise 12.7:**

What are the implications of bypassing/short-circuiting on the length of combinational paths in a design, and the consequent effect on achievable clock frequency?

□

---

An even more advanced form of bypassing (with much more circuit complexity) would be:

- Eliminate the scoreboard; do not stall an instruction in the Register-Read stage, but allow it to move into its appropriate Execute stage, and stall it there if necessary. This frees up the Register-Read stage to process the next instruction, which may move into a different Execute stage.
- When Retire writes a register value, broadcast it to the different Execute stages to enable instructions there that are stalled on this register value.

## 12.4 Retiring outputs of the Execute Stages in Order with Tags

In Figure 12.1, each yellow box in the Execute stage is an independent pipeline handling a certain subset of the instruction set. For example, "Execute Control" handles BRANCH, JAL and JALR instructions. "Execute Integer Arithmetic and Logic Ops" handles LUI, AUIPC, and all arithmetic and logic instructions. "Exec Mem Op" handles LOAD and STORE instructions. If we extend Fife to handle the "M" ISA extension, we would have a pipeline for integer multiply and divide instructions. If we extend Fife to handle the "F" and "D" ISA extension, we would have a pipeline for floating point arithmetic. The Register-Read-and-Dispatch stage sends information into these pipes depending on the kind of instruction.

Instructions may have different latencies in traversing these Execute pipes. For example, Control and Integer ops may typically traverse in one clock, but multiplication, division, floating point and memory ops may take more clocks. The latency variation may be data dependent: for example multiplication/division may recognize the special case where an operand is 0 or 1 and return a result quickly. A memory op may return quickly on a cache hit, and take more time on a cache miss.

The Retire stage needs to gather the outputs from the Execute stages and retire them in the proper order. But, because of varying latency, availability of data is not an indication of the proper order.

The solution to this "ordering" problem is *tags*. In Figure 12.1 we see there is also a *direct path* from Register-Read-and-Dispatch to Retire. We pass a tag on this path *for every instruction*. For example if the instruction is a BRANCH instruction, the Register-Read-and-Dispatch stage sends information into Execute Control, but it also sends a tag `EXEC_TAG_CONTROL` on the direct path to Retire, indicating that it has just dispatched an instruction into Execute Control.

Thus, the sequence of tags on the direct path tells Retire exactly the order in which to service the various Execute pipes. Retire always looks at tag on the direct path first. For example, if Retire sees a `EXEC_TAG_DMEM` tag on the direct path, it knows that it must next look for an output from the Exec Memory Ops pipe, even if outputs are already available on the Execute Control and/or Execute Integer pipes from later instructions.

## 12.5 Allowing Memory Ops to be Pipelined, with a Store Buffer

Consider the Execute Memory Ops stage in Figure 12.1, which issues LOAD and STORE requests to memory and collects their responses. At this stage, the instruction is still speculative; it may be discarded when it reaches the Retire stage. We must ensure that STORE instructions do not yet modify memory permanently.

The mechanism for this purpose is the "*store buffer*" shown Figure 12.1. This is a buffer in front of the memory system (between the CPU and the memory system).

- The store buffer is itself a queue.
- When we execute a STORE instruction, the address, data and size are appended to the STORE buffer queue. When the instruction finally reaches the Retire stage, the Retire stage sends a final "commit/discard" message to the store-buffer. If a commit,

the STORE at the head of the store-buffer is committed to memory and we dequeue it from the store-buffer. If a discard, we just dequeue it from the store-buffer.
- When we execute a LOAD instruction, we first check the store-buffer if there have been any recent updates to the address in question, and then go to the memory system behind it if necessary.

### 12.5.1 What about LOADs and STOREs to non-memory-like devices (MMIO)?

In RISC-V there are no separate instructions for input and output to devices. Devices contain "device registers" which are placed at particular "memory addresses" and are accessed from the CPU just like memory, with LOAD and STORE instructions. We say that the device registers are "mapped" to those addresses. These accesses can control and configure the device, and move data between the CPU and the device. Such a scheme is known as MMIO, for Memory-Mapped Input-Output.

Device registers, although addressed like memory locations, may behave quite differently from memory, in several ways:

- LOADs may have side-effects. For example, a LOAD from a memory location does not (observably) disturb anything, but a LOAD from a device could switch on an LED, or increment a counter.
- LOADs may not be idempotent. For example, two successive LOADs from a memory location return the same value, whereas two successive LOADs from, say, a UART's "receiver buffer register" may return two successive (and different) characters from a keyboard.
- STOREs may have additional side-effects. A STORE to a memory location merely stores the value there. A STORE to a device register may display it on a screen, start a motor, release a wheel brake, and so on.
- In memory, a LOAD returns the value stored there by the most recent STORE. For a UART device, on the other hand, a STORE may display a character on a screen whereas a subsequent LOAD from the same address may return a character from the keyboard.

For these reasons, it is dangerous to perform any LOAD or STORE speculatively on non-memory devices. The "Execute Memory Ops" stage does not even attempt the access; it simply defers such a request for future execution by the Retire stage. The decision whether to defer a request or not is based on the address.

Once the LOAD/STORE instruction has reached the Retire stage and we know for sure that it is not speculative, the Retire stage performs the memory operation: it sends the request to memory, collects the response, and retires the instruction.

---

**Exercise 12.8:**

When the Retire stage sends a "commit/discard" message, how do we know that it applies to the pending STORE at the head of the store-buffer queue?

**Exercise 12.9:**

For a speculative LOAD, its value may come from memory or from the store-buffer. In what order should it scan these sources?

**Exercise 12.10:**

Does the Retire stage need to send both "commit" and "discard" messages? If requests are accompanied by an instruction-number, the Retire stage could only send "commit" messages accompanied by the instruction-number. The store-buffer can discard all pending STOREs with earlier instruction numbers.

Discuss the implications of such a design on the size of the store-buffer.

□

---

## 12.6 The Retire Stage

Figure 12.2 summarizes the actions to be taken by the Retire stage.

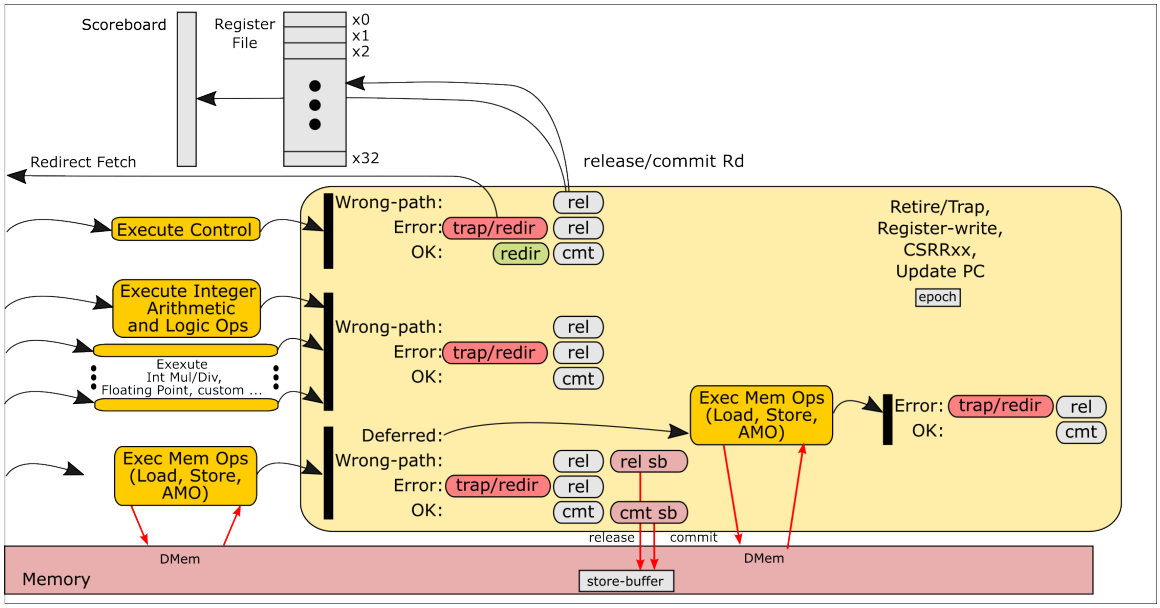



Figure 12.2: Actions in the "Retire" stage of Fife

Any of the Execute pipes can produce a wrong-path instruction (accompanying epoch does not match current epoch). In each such case, we discard the instruction, but if the instruction has taken a scoreboard reservation, we must also release that ("rel" annotations). In the case of LOAD/STORE that was performed by the Exec Memory Ops stage, we must also send a "discard" message to the store-buffer.

Any of the Execute pipes can produce an exception. In each case we must redirect the Fetch stage to the trap-handler PC, but before we do that we must release scoreboard reservations, if any, and send a "discard" message to the store-buffer, if necessary.

If the output of the Execute pipes is "OK" (not wrong-path, not an exception), then we can retire the instruction. Control instructions may redirect the Fetch stage. Control, IALU and LOAD instructions may write a register value (and release its reservation). STORE instructions must send a "commit" message to the store-buffer.

Finally, deferred LOAD/STORE instructions must be excuted, sending a request to memory, collecting its response and retiring it.

Retire actions in Fife

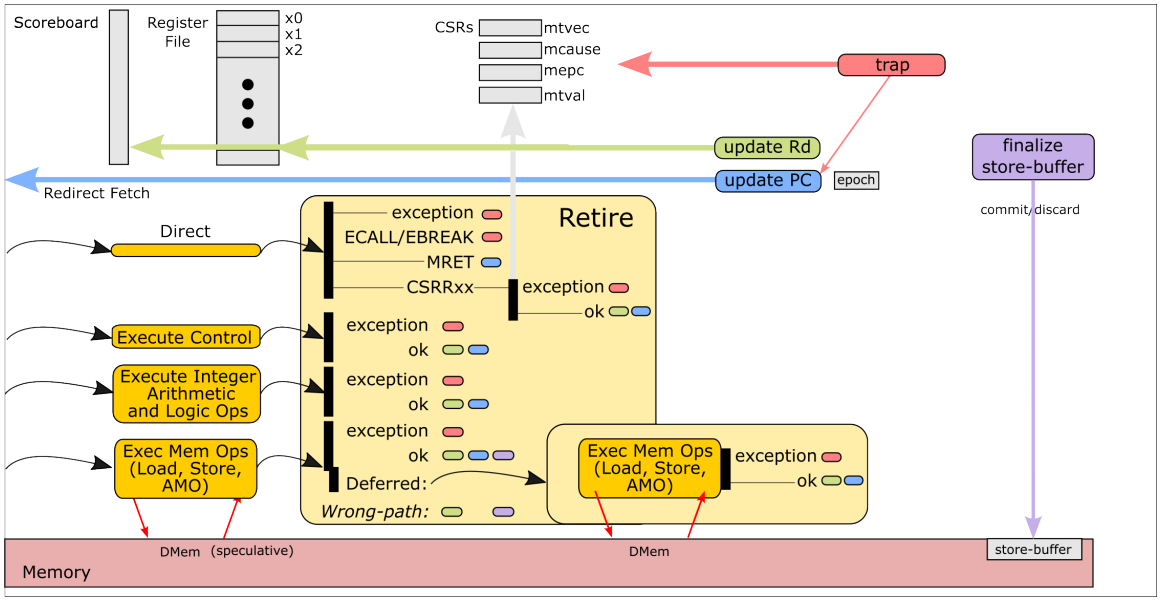



Figure 12.3: Retire actions in Fife

# Chapter 13

# RISC-V: the Fife pipelined CPU: BSV code

## 13.1 Introduction

In this chapter we study BSV code to implement the principles that were discussed in the previous chapter. We repeat Figure 12.1 here, for reference.

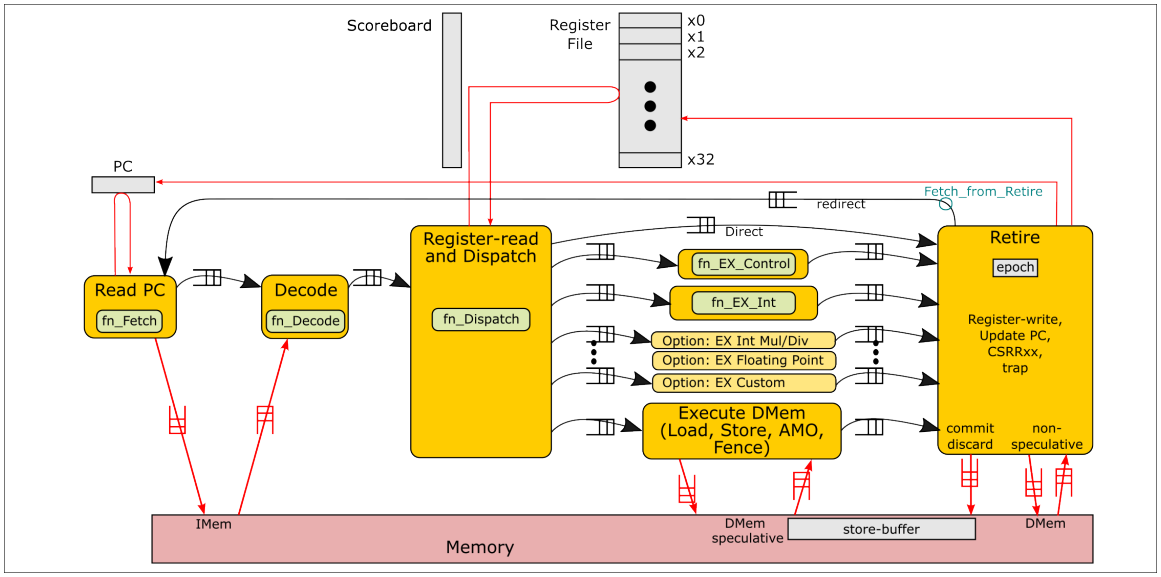



Figure 13.1: Pipelined interpretation of RISC-V instructions (Fig. 3.1 with some annotations)

## 13.2 The Fife top-level CPU module

The code for the top-level Fife CPU module is actually simpler than the code for the Drum CPU module:

(In file:src_Fife/CPU.bsv)

```
(* synthesize *)
module mkCPU (CPU_IFC);
   // Pipeline stages
   Fetch_IFC       stage_F          <- mkFetch;
   Decode_IFC      stage_D          <- mkDecode;
   RR_RW_IFC       stage_RR_RW      <- mkRR_RW;
   EX_Control_IFC  stage_EX_Control <- mkEX_Control;  // Branch, JAL, JALR
   EX_Int_IFC      stage_EX_Int     <- mkEX_Int;      // Integer ops
   Retire_IFC      stage_Retire     <- mkRetire;

   // ================================================================
   // Forward flows

   // F->D->RR_RW->Retire
   mkConnection (stage_F.fo_F_to_D,           stage_D.fi_F_to_D);
   mkConnection (stage_D.fo_D_to_RR,          stage_RR_RW.fi_D_to_RR);
   mkConnection (stage_RR_RW.fo_RR_to_Retire, stage_Retire.fi_RR_to_Retire);

   // RR->various EX
   mkConnection (stage_RR_RW.fo_RR_to_Control, stage_EX_Control.fi_RR_to_Control);
   mkConnection (stage_RR_RW.fo_RR_to_EX_IALU, stage_EX_Int.fi_RR_to_EX_IALU);

   // Various EX->Retire
   mkConnection (stage_EX_Control.fo_Control_to_Retire,
                 stage_Retire.fi_Control_to_Retire);
   mkConnection (stage_EX_Int.fo_EX_IALU_to_Retire,
                 stage_Retire.fi_IALU_to_Retire);

   // ================================================================
   // Backward flows

   // F<-RR (redirection)
   mkConnection (stage_Retire.fo_F_from_Retire,  stage_F.fi_F_from_Retire);
   // RR<-RW (writeback/trap/flush)
   mkConnection (stage_Retire.fo_RW_from_Retire, stage_RR_RW.fi_RW_from_Retire);

   // ================================================================
   // INTERFACE

   method Action init (Initial_Params initial_params);
      stage_F.init (initial_params);
      ... and similarly, for all the other stages ...
   endmethod

   interface fo_IMem_req = stage_F.fo_F_to_IMem;
   interface fi_IMem_rsp = stage_D.fi_IMem_to_D;

   interface fo_DMem_S_req    = stage_RR_RW.fo_DMem_S_req;
   interface fi_DMem_S_rsp    = stage_Retire.fi_DMem_S_rsp;
   interface fo_DMem_S_commit = stage_Retire.fo_DMem_S_commit;

   interface fo_DMem_NS_req = stage_Retire.fo_DMem_NS_req;
   interface fi_DMem_NS_rsp = stage_Retire.fi_DMem_NS_rsp;
```

```
54 endmodule
```

This is practically a direct textual description of Figure 13.1. The first few lines instantiate the pipeline stages shown in the figure. There is no explicit module corresponding to Execute Memory Ops—the DMem request is sent out directly from `stage_RR_RW` and the DMem response is collected directly by `stage_Retire`.

The next several lines make the "forward-flow" connections between modules (left to right in the figure) . These are followed by lines making the "backward-flow" connections. All these use the `mkConnection` module to connect a `FIFOF_O` interface (producer) to a `FIFOF_I` interface (consumer), which was discussed in Section 7.5.6.

In the INTERFACE section, after the `init` method, the next two lines are the flows of IMem requests from the Fetch stage to memory and IMem responses from memory to the Decode stage. These just lift interfaces from `stage_F` and `stage_D` to the CPU interface, as is.

The next three lines are for *speculative* DMem access, which we discussed in Section 12.5: the flow of DMem requests from `stage_RR` to memory, the flow of DMem responses from memory to `stage_Retire`, and the flow of "commit/discard" messages from `stage_Retire` to the store-buffer to discharge STOREs that are waiting in the store-buffer.

The last two lines are for *non-speculative* DMem access, which we discussed in Section 12.5.1.

Note that the module interface `CPU_IFC` is exactly the same as in Drum (although Drum has no need for, and does not use the speculative DMem interfaces). Thus, in a system context, we can directly substitute Drum for Fife and vice versa. Generalizing, we can develop other CPUs and substitute them, as well.

## 13.3 Fife: the Fetch stage

The Fetch stage module code is shown below.

(In file:src_Fife/S1_Fetch.bsv)

```
1  (* synthesize *)
2  module mkFetch (Fetch_IFC);
3     Reg #(Bool) rg_running <- mkReg (False);
4
5     // Forward out
6     FIFOF #(F_to_D)  f_F_to_D    <- mkBypassFIFOF;
7     FIFOF #(Mem_Req) f_F_to_IMem <- mkBypassFIFOF;
8
9     // Backward in
10    FIFOF #(F_from_Retire) f_F_from_Retire <- mkPipelineFIFOF;
11
12    // inum, PC and epoch registers
13    Reg #(Bit #(64))       rg_inum  <- mkReg (0);
14    Reg #(Bit #(XLEN))     rg_pc    <- mkReg (0);
15    Reg #(Bit #(W_Epoch))  rg_epoch <- mkReg (0);
16
17    // ----------------------------------------------------------------
18    // BEHAVIOR
```

```
19
20    // Forward flow
21    rule rl_Fetch_req (rg_running && (! f_F_from_Retire.notEmpty));
22       // Predict next PC
23       let pred_pc = rg_pc + 4;
24
25       let y <- fn_F (rg_inum, rg_pc, pred_pc, rg_epoch);
26       f_F_to_D.enq (y.to_D);
27       f_F_to_IMem.enq (y.mem_req);
28
29       rg_pc   <= pred_pc;
30       rg_inum <= rg_inum + 1;
31    endrule
32
33    // Backward flow: redirection from Retire
34    rule rl_F_from_Retire;
35       let x <- pop_o (to_FIFOF_O (f_F_from_Retire));
36
37       rg_pc    <= x.next_pc;
38       rg_epoch <= x.next_epoch;
39    endrule
40
41    // ----------------------------------------------------------------
42    // INTERFACE
43
44    method Action init (Initial_Params initial_params) if (! rg_running);
45       ...
46       rg_pc      <= initial_params.pc_reset_value;
47       rg_running <= True;
48    endmethod
49
50    // Forward out
51    interface fo_F_to_D    = to_FIFOF_O (f_F_to_D);
52    interface fo_F_to_IMem = to_FIFOF_O (f_F_to_IMem);
53
54    // Backward in
55    interface fi_F_from_Retire = to_FIFOF_I (f_F_from_Retire);
56 endmodule
```

The first section of the module instantiates some registers and FIFOs.

Next, the BEHAVIOR section contains two *rules*, which are the fundamental dynamic behavior constructs in BSV. A rule is an infinite process. Every rule consists of a *condition* and an *action*: whenever the condition is true, the action is performed; we say the rule "fires" whenever its condition is true.[1]

A rule may contain explicit and implicit conditions. In rule `rl_Fetch_req`, the explicit condition is the expression:

```
(rg_running && (! f_F_from_Retire.notEmpty))
```

Rule `rl_Fetch_req`'s implicit conditions come from the methods that it invokes, specifically: `f_F_to_D.enq ()` and `f_F_to_IMem.enq ()`. Every method has an implicit conditions. For a FIFO, the *enq()* method's

[1] Topic discussed later in the book: a rule may not fire even if its condition is true, if it "conflicts" with another rule.

implicit condition is false when the FIFO is full, *i.e.,* when it does not contain space to enqueue a new item. We also say the method is "enabled" when its implicit condition is true.

In summary, rule `rl_Fetch_req` will fire only when the explicit condition is true, and when both FIFOs on which it tries to enqueue are enabled. When it fires, it performs a composite action that includes all the actions stated in the rule:

- enqueues into `f_F_to_D` the value of `y.to_D`,
- enqueues into `f_F_to_IMem` the value of `y.mem_req`,
- writes `pred_pc` into `rg_pc`,
- and writes the value of `rg_inum+1` into `rg_inum`.

where `y` is the result of applying `fn_F` to `rg_inum`, `pred_pc` and `rg_epoch`, where `pred_pc` is the value of `rg_pc+4.`,

NOTE:

> The function `fn_F()` invoked in rule `rl_Fetch` is exactly the same one as `fn_F()` used in the Fetch step of Drum.
>
> The types of the messages passed to the Decode stage (`y.to_D` of type `F_to_D`) and to memory (`y.mem_req` of type `Mem_Req`) are the same as in Drum.

All these actions are semantically *instantaneous* and *simultaneous*. Note that when the rule's implicit and explicit conditions are true, all the actions are peformed; if false, none of them are performed. We also say that the rule's composite action is "atomic".

In summary, rule `rl_Fetch_req` computes an IMem memory request from the PC and sends it to memory; it sends auxiliary information to the Decode stage; and it updates the PC and inum in preparation for the next Fetch.

Rule `rl_F_from_Retire` receives, in `x`, a redirection message from the Retire stage, and updates the PC and epoch accordingly. This rule has no explicit conditions; its single implicit condition comes from `f_F_from_Retire`'s implicit condition that we cannot pop a value from the FIFO if it is empty, *i.e.,*, this rule only fires when a redirection message is available. When it fires, it performs three actions atomically/instantaneously/simultaneously:

- It dequeues `x` from the FIFO `f_F_from_Retire` (the dequeue action is inside the `pop_o` function),
- It updates `rg_pc` with the new PC in the redirection message,
- It updates `rg_epoch` with the new epoch in the redirection message.

Note, `rl_F_from_Retire` updates two registers `rg_pc` and `rg_epoch` and, *concurrently*, `rl_Fetch` reads both those registers. Because rule actions are atomic, we are guaranteed that `rl_Fetch` will not see inconsitent values in those two registers, where one has been updated but the other has not yet been updated.

Finally, the INTERFACE section of the module is simple. After the `init` method, we simply lift the FIFO interfaces to the `mkFetch` interface.

## 13.4 Fife: the Decode stage

The Decode stage module code is shown below.

(In file:src_Fife/S2_Decode.bsv)

```
1 (* synthesize *)
2 module mkDecode (Decode_IFC);
3    // Forward flows in
4    // Depth should be > F=>IMem=>D path latency
5    FIFOF #(F_to_D)  f_F_to_D    <- mkSizedFIFOF (4);
6    FIFOF #(Mem_Rsp) f_IMem_to_D <- mkPipelineFIFOF;
7
```

```
8     // Forward flow out
9     FIFOF #(D_to_RR) f_D_to_RR   <- mkBypassFIFOF;
10
11    // ================================================================
12    // Functionality
13
14    rule rl_Decode;
15       F_to_D  x        <- pop_o (to_FIFOF_O (f_F_to_D));
16       Mem_Rsp rsp_IMem <- pop_o (to_FIFOF_O (f_IMem_to_D));
17
18       D_to_RR y <- fn_D (x, rsp_IMem);
19
20       f_D_to_RR.enq (y);
21    endrule
22
23    // ================================================================
24    // INTERFACE
25
26    method Action init (Initial_Params initial_params);
27       ...
28    endmethod
29
30    // Forward flows in
31    interface fi_F_to_D    = to_FIFOF_I (f_F_to_D);
32    interface fi_IMem_to_D = to_FIFOF_I (f_IMem_to_D);
33    // Forward flows out
34    interface fo_D_to_RR = to_FIFOF_O (f_D_to_RR);
35 endmodule
```

The first section instantiates FIFOs for incoming and outgoing flows.

The single rule `rl_Decode`'s implicit conditions will make it wait for both incoming FIFOs `f_F_to_D` and `f_IMem_to_D` to be non-empty. When the rule fires, it:

- pops `x` and `rsp_Mem` from the two FIFOs, respectively,
- applies function `fn_D()` to those values (this is the *same* `fn_D()` that was used in the Decode step of Drum), and
- sends the result on to the Register-Read stage.

The INTERFACE section is again straightforward, just lifting the FIFO interfaces to this module's interface.

## 13.5 Fife: RR-RW module (Register-Read, Dispatch, Register-Write)

The next module ("RR-RW") contains the register file and the scoreboard, and performs several functions. In the forward flow ("Register-Read and Dispatch" stage):

- stall (wait) if the instruction has rs1, rs2 or rd, and these are busy according to the scoreboard;
- read rs1 and rs2 registers (if needed) for the current instruction;
- set the scoreboard for rd to 1, meaning "busy" (if the instruction has an rd);
- use the information from the Decode stage to dispatch to the Execute pipes. We always (for every instruction) send an execution tag and additional information on the direct path to the Retire stage. We possibly send information into one of the following Execute pipes:
  - to Execute Control stage, or

- to Execute Control stage, or
- to memory (a DMem request).

In the backward flow ("Register-Write stage"), it receives messages from the Retire stage to release scoreboard reservations (for instructions that have retired or aborted due to mispredictions or traps) and to write-back rd values of retired instructions.

### 13.5.1 BSV: Vectors for the Scoreboard

In Section 12.3 we discussed the general principles of a scoreboard, and described it as an array of 1-bit values. In BSV the following type is used to represent an array of $n$ items, each of which is of type $t$:

```
Vector #(n, t);
```

Note: in order to use this type in any BSV code file, the file must import the `Vector` library:

```
import Vector ::  *;
```

So, we can define a `Scoreboard` type as follows:

```
typedef Vector #(32, Bit #(1)) Scoreboard;
```

The BSV Vector-library function:

```
replicate (v)
```

creates a value of type `Vector #(n, t)` where $n$ is inferred from the context and `v` has the type $t$. All $n$ items in the value are equal to `v`. Thus, we can instantiate a scoreboard like this, where all the vector elements are initialized to 0:

```
Reg #(Scoreboard) rg_scoreboard <- mkReg (replicate (0));
```

In hardware, a `Vector#(32,Bit#(1))` value occupies exactly 32 bits, *i.e.,* the size of the vector times the size of each element. So, why not use `Bit #(32)` instead? It's a matter of programming taste:

- The same syntax `v[j]` works both for bit-selection from `Bit#(n)` and `Vector#(n,Bit#(1))`.
- With `Bit#(n)`, a $j^{th}$ bit can also be selected using shift-and-mask operations: `((v >> j) & 1)`.
- The $j^{th}$ bit of `Vector#(n,Bit#(1))` can be updated using simple assignment
  `v [j] = new_value;`
- The $j^{th}$ bit of `Bit#(n)` can be updated using shift and mask operations:
  `(v | (1 << j))` to set the $j^{th}$ bit to 1, and
  `(v & (~ (1 << j)))` to resset the $j^{th}$ bit to 0

We can convert a `Vector #(32, Bit #(1))` value into a `Bit#(32)` value with:

```
pack (v)
```

and we can convert a `Bit#(32)` value into a `Vector #(32, Bit #(1))` value with:

```
unpack (v)
```

### 13.5.2 The RR-RW module

The RR-RW module code is shown below (except we elide the BEHAVIOR rules, which we will discuss shortly.

(In file:src_Fife/S3_RR_RW.bsv)

```
(* synthesize *)
module mkRR_RW (RR_RW_IFC);
   // Forward in
   FIFOF #(D_to_RR) f_D_to_RR <- mkPipelineFIFOF;

   // Forward out
   FIFOF #(RR_to_Retire)  f_RR_to_Retire  <- mkBypassFIFOF;  // Direct
   FIFOF #(RR_to_Control) f_RR_to_Control <- mkBypassFIFOF;  // Control pipe
   FIFOF #(RR_to_EX)      f_RR_to_EX_IALU <- mkBypassFIFOF;  // Integer pipe
   FIFOF #(Mem_Req)       f_DMem_S_req    <- mkBypassFIFOF;  // DMem pipe

   // Backward in
   FIFOF #(RW_from_Retire) f_RW_from_Retire <- mkPipelineFIFOF;

   // The integer register file (GPRs)
   RISCV_GPRs_IFC #(XLEN)  gprs <- mkRISCV_GPRs;

   // Scoreboard for GPRs
   Reg #(Scoreboard) rg_scoreboard <- mkReg (replicate (0));

   // ================================================================
   // BEHAVIOR

   ... (to be discussed shortly) ...

   // ================================================================
   // INTERFACE

   method Action init (Initial_Params initial_params);
      ...
   endmethod

   // Forward in
   interface fi_D_to_RR = to_FIFOF_I (f_D_to_RR);

   // Forward out
   interface fo_RR_to_Retire  = to_FIFOF_O (f_RR_to_Retire);
   interface fo_RR_to_Control = to_FIFOF_O (f_RR_to_Control);
   interface fo_RR_to_EX_IALU = to_FIFOF_O (f_RR_to_EX_IALU);
   interface fo_DMem_S_req    = to_FIFOF_O (f_DMem_S_req);

   // Backward in
   interface fi_RW_from_Retire = to_FIFOF_I (f_RW_from_Retire);
endmodule
```

The first section instantiates FIFOs `f_XXX` for all the forward and backward flows, and instantiates the register file `gprs` and the `scoreboard`.

The final INTERFACE section, after the `init` method, simply lists the FIFO interfaces to this module's interface.

The BEHAVIOR section contains two rules, one for the forward flow (Register-Read and Dispatch) and one for the backward flow (Register-Write). The forward-flow rule is shown below:

(In file:src_Fife/S3_RR_RW.bsv)

```
rule rl_RR_Dispatch;
   let x = f_D_to_RR.first;

   let instr   = x.instr;
   let opclass = x.opclass;
   let rs1     = instr_rs1 (instr);
   let rs2     = instr_rs2 (instr);
   let rd      = instr_rd  (instr);

   let scoreboard = rg_scoreboard;
   let busy_rs1   = (x.has_rs1 && scoreboard [rs1]);
   let busy_rs2   = (x.has_rs2 && scoreboard [rs2]);
   let busy_rd    = (x.has_rd  && scoreboard [rd]);
   Bool stall     = (busy_rs1 || busy_rs2 || busy_rd);

   if (stall) begin
      // no action
   end
   else begin
      // Not stalled; proceed
      f_D_to_RR.deq;

      // Read GPRs
      let rs1_val = gprs.read_rs1 (rs1);
      let rs2_val = gprs.read_rs2 (rs2);

      // Dispatch to one of the next-stage pipes
      Result_Dispatch y <- fn_Dispatch (rg_flog, x, rs1_val, rs2_val);

      // Direct to Retire
      f_RR_to_Retire.enq (y.to_Retire);

      // Dispatch
      if (y.to_Retire.exec_tag == EXEC_TAG_RETIRE) begin
         // no further action
      end
      else begin
         // Update scoreboard for RD
         if (x.has_rd) begin
            scoreboard [rd] = 1;
            rg_scoreboard <= scoreboard;
         end

         // Forward info to appropriate Execute pipe
         if (y.to_Retire.exec_tag == EXEC_TAG_CONTROL)
            f_RR_to_Control.enq (y.to_Control);

         else if (y.to_Retire.exec_tag == EXEC_TAG_IALU)
```

```
49              f_RR_to_EX_IALU.enq (y.to_EX);
50
51           else if (y.to_Retire.exec_tag == EXEC_TAG_DMEM) begin
52              Mem_Req mem_req <- fn_DMem_Req (rg_flog, y.to_EX);
53              f_DMem_S_req.enq (mem_req);
54           end
55           else begin
56              $display ("    -> IMPOSSIBLE");
57              $finish (1);
58           end
59        end
60     end
61  endrule
```

In line 2, we observe the first element in the `f_D_to_RR` FIFO. Note, we observe it non-destructively, *i.e., we do not dequeue* it from the FIFO. This is because, if we must stall, it needs to be available the next time the rule fires.

The next several lines compute the stall condition by checking the scoreboard for whether rs1, rs2 or rd are busy (if the instruction has rs1, rs2 or rd, respectively).

If we stall, the rule takes no action; everything will be retried the next time it fires. If we do not stall, then we dequeue the `f_D_to_RR` FIFO. We read values of the rs1 and rs2 registers. Note, if the instruction does not have an rs1 or rs2, here we will be reading some random registers according to the bits that happen to be in the rs1 and rs2 bit-positions in the instruction. This does not matter; in the Execute stage each instruction only *uses* these values if the instruction has an rs1 and/or rs2.

Next, we apply the function `fn_Dispatch` (it was discussed in Section 6.4, and is the same one we use in Drum) to the inputs, which computes `y`, containing the structs to be sent on the direct path (`y.to_Retire`), to Execute Control (`y.to_Control`), and to Execute Int and Execute DMem (`y.toEX`).

We enqueue `y.to_Retire` on the direct flow (FIFO `f_RR_to_Retire`). If the execution tag is not `EXEC_TAG_RETIRE`, we will be sending the instruction into the one of the Execute pipes (Execute Control, Execute Integer, or DMem). If the instruction has an rd, we mark it on the scoreboard, and send the instruction into the appropriate pipe.

---

**Exercise 13.1:**

Consider this hypothetical scenario: suppose the `stall` condition is false. Then, we need to dequeue `f_D_to_RR` and do one or more enqueues into `f_RR_to_Retire` and `f_RR_to_Control`, `f_RR_to_EX_IALU` or `f_DMem_S_req.enq`. Is it possible that we perform the dequeue and then are unable to perform the enqueue(s) because the corresponding output FIFO happens to be full?

*Hint:* Remember the definition of rule atomicity where a rule only fires if the implicit conditions on *all* the methods it invokes are true.

**Exercise 13.2:**

Write a boolean expression representing the overall firing condition for the rule. Briefly: all FIFO-modifying actions (dequeue, enqueue) have implicit conditions, but for each FIFO, that condition is only relevant if the conditions on the surrounding if-then-else's select that action.

**Exercise 13.3:**

When debugging the implementation, it is useful to know if, due to some coding mistake, rule `rl_RR_Dispatch` is stalled forever. For example, for some instruction with an rd, if the Retire stage did not send back the rd's scoreboard-release, that register will be forever "busy".

Add a register to count consecutive stalls, initially 0. In the rule, whenever we successfully dispatch an instruction, reset the counter to 0. Whenever we stall, increment the stall counter, and if the stall-counter reaches some chosen threshold value, prints debugging messages and executes `$finish` to terminate simulation.

□

---

Here is the code for rule `rl_RW_from_Retire` For the backward flow.

(In file:src_Fife/S3_RR_RW.bsv)

```
1   rule rl_RW_from_Retire;
2      let x <- pop_o (to_FIFOF_O (f_RW_from_Retire));
3
4      Scoreboard scoreboard = rg_scoreboard;
5      scoreboard [x.rd] = 0;
6      rg_scoreboard <= scoreboard;
7
8      if (x.commit)
9         gprs.write_rd (x.rd, x.data);
10  endrule
```

We pop the message `x` from the `f_RW_from_Retire` FIFO. We perform its specified scoreboard-release for register rd. If the rd value is to be committed, we write it into GPR [rd].

## 13.6 Fife: the Execute Control stage

The code for the Execute Control stage is shown below:

(In file:src_Fife/S4_EX_Control.bsv)

```
1   (* synthesize *)
2   module mkEX_Control (EX_Control_IFC);
3      // Forward in
4      FIFOF #(RR_to_Control)      f_RR_to_Control      <- mkPipelineFIFOF;
5      // Forward out
6      FIFOF #(Control_to_Retire)  f_Control_to_Retire  <- mkBypassFIFOF;
7
8      // ================================================================
9      // BEHAVIOR
10
11     rule rl_Control;
12        let x <- pop_o (to_FIFOF_O (f_RR_to_Control));
13        let y <- fn_Control (rg_flog, x);
14        f_Control_to_Retire.enq (y);
15     endrule
16
17     // ================================================================
18     // INTERFACE
19
20     method Action init (Initial_Params initial_params);
21        ...
22     endmethod
23
24     // Forward in
```

```
25    interface fi_RR_to_Control     = to_FIFOF_I (f_RR_to_Control);
26    // Forward out
27    interface fo_Control_to_Retire = to_FIFOF_O (f_Control_to_Retire);
28 endmodule
```

After instantiating the forward flow input and output FIFOs, the rule `rl_Control` simply applies the function `fn_Control` to each input and enqueues the output. This function was described in Section 6.5, and is the same one we use in Drum. Finally, the interface, after the `init` method, simply lifts the FIFO interfaces to the interface of this module.

## 13.7 Fife: the Execute Integer Ops stage

The code for the Execute Integer Ops stage is also very simple, and shown below:

(In file:src_Fife/S4_EX_Int.bsv)

```
1  (* synthesize *)
2  module mkEX_Int (EX_Int_IFC);
3     // Forward in
4     FIFOF #(RR_to_EX)      f_RR_to_EX_IALU <- mkPipelineFIFOF;
5     // Forward out
6     FIFOF #(EX_to_Retire)  f_EX_to_Retire  <- mkBypassFIFOF;
7
8     // ================================================================
9     // BEHAVIOR
10
11    rule rl_EX_IALU;
12       let x <- pop_o (to_FIFOF_O (f_RR_to_EX_IALU));
13       let y <- fn_EX_IALU (rg_flog, x);
14       f_EX_to_Retire.enq (y);
15    endrule
16
17    // ================================================================
18    // INTERFACE
19
20    method Action init (Initial_Params initial_params);
21       ...
22    endmethod
23
24    // Forward in
25    interface fi_RR_to_EX_IALU = to_FIFOF_I (f_RR_to_EX_IALU);
26
27    // Forward out
28    interface fo_EX_IALU_to_Retire = to_FIFOF_O (f_EX_to_Retire);
29 endmodule
```

The structure is just like `mkEX_Control`: forward-flow input and output FIFOs, with the rule transforming input to output through the function `fn_EX_IALU`, which was discussed in Section 6.6 and is the same one we use in Drum. Finally, the interface, after the `init` method, simply lifts the FIFO interfaces to the interface of this module.

## 13.8 Fife: the Execute Memory Ops stage (speculative DMem)

There is no explicit code for an Execute Memory Ops stage. The forward-path rule `rl_RR_Dispatch` in the Register-Read-and-Dispatch stage, described in Section 13.5, directly enqueues a memory request that goes out to memory. The Retire stage, to be discussed in Section 13.9, consumes the corresponding memory response.

## 13.9 Fife: the Retire stage

This module is longer than the others only because it takes care of many possible cases as outlined in Figure 13.2 (which repeats Figure 12.2 here for reference).

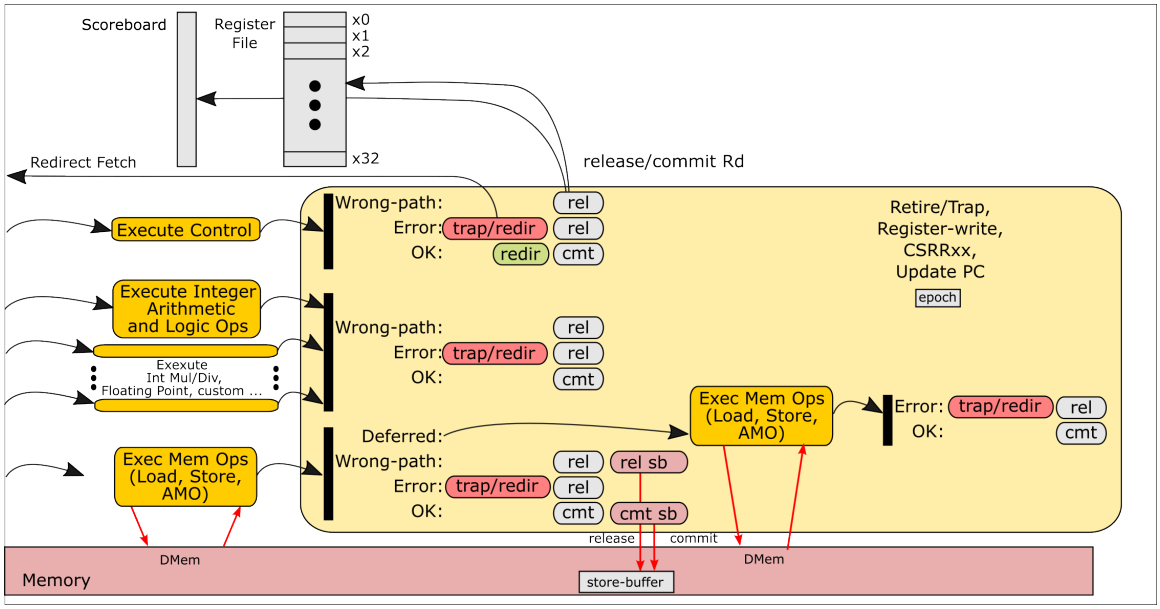



Figure 13.2: Actions in the "Retire" stage of Fife (same as Fig. 12.2)

Here is the interface definition for this module:

(In file:src_Fife/S5_Retire.bsv)

```
interface Retire_IFC;
   method Action init (Initial_Params initial_params);

   // Forward in
   interface FIFOF_I #(RR_to_Retire)       fi_RR_to_Retire;
   interface FIFOF_I #(Control_to_Retire)  fi_Control_to_Retire;
   interface FIFOF_I #(EX_to_Retire)       fi_IALU_to_Retire;

   // Speculative DMem response and commit/discard
   interface FIFOF_I #(Mem_Rsp)                fi_DMem_S_rsp;
   interface FIFOF_O #(Retire_to_DMem_Commit)  fo_DMem_S_commit;

   // Non-speculative DMem
   interface FIFOF_O #(Mem_Req)  fo_DMem_req;
   interface FIFOF_I #(Mem_Rsp)  fi_DMem_rsp;

```

```
17    // Backward out
18    interface FIFOF_O #(F_from_Retire)      fo_F_from_Retire;
19    interface FIFOF_O #(RW_from_Retire)     fo_RW_from_Retire;
20 endinterface
```

The first four `fi_XXX` sub-interfaces correspond to the black arrows entering the Retire module from the left in the figure. The next two `fo_XXX` sub-interfaces correspond to the outgoing red arrows at the bottom of the module, and the next sub-interface (`fi_DMem_rsp`) is the incoming red arrow at the bottom of the module returning a non-speculative memory response. The last two sub-interfaces are the black output arrows at the top of the module carrying redirections to the Fetch stage and Register-Writes to the `RR_RW` module, respectively.

We define a "state" for this module:

(In file:src_Fife/S5_Retire.bsv)

```
1  typedef enum {STATE_PIPE,        // Normal pipeline operation
2                STATE_DMEM_RSP     // Handle Non-speculative DMem response
3  } Module_State
4  deriving (Bits, Eq, FShow);
```

Normally the module is in `STATE_PIPE`, acting as the last stage of the pipeline. However, in certain circumstances we switch into non-pipelined "FSM" mode, similar to Drum:

- to execute non-speculative DMem ops (MMIO/non-memory-like),
- to execute traps and interrupts (discussed in a later chapter), and
- to execute CSRRx insructions (discussed in a later chapter).

Here we see the code for the `mkRetire` module, the Retire stage (we have elided the BEHAVIOR section which will be discussed shortly).

(In file:src_Fife/S5_Retire.bsv)

```
1  (* synthesize *)
2  module mkRetire (Retire_IFC);
3     // For managing speculation, redirection traps, etc.
4     Reg #(Epoch) rg_epoch  <- mkReg (0);
5
6     // Forward in
7     // Depth of f_RR_to_Retire should be > longest EX pipe
8     FIFOF #(RR_to_Retire)       f_RR_to_Retire      <- mkSizedFIFOF (8);
9     FIFOF #(Control_to_Retire)  f_Control_to_Retire <- mkPipelineFIFOF;
10    FIFOF #(EX_to_Retire)       f_IALU_to_Retire    <- mkPipelineFIFOF;
11    FIFOF #(Mem_Rsp)            f_DMem_S_rsp        <- mkPipelineFIFOF;
12
13    // Forward out
14    FIFOF #(Retire_to_DMem_Commit) f_DMem_S_commit  <- mkBypassFIFOF;
15
16    // Backward out
17    FIFOF #(F_from_Retire)   f_F_from_Retire  <- mkBypassFIFOF;
18    FIFOF #(RW_from_Retire)  f_RW_from_Retire <- mkBypassFIFOF;
19
20    // Non-speculative DMem reqs and rsps
21    FIFOF #(Mem_Req)  f_DMem_req  <- mkBypassFIFOF;
22    FIFOF #(Mem_Rsp)  f_DMem_rsp  <- mkPipelineFIFOF;
23
24    Reg #(Module_State) rg_state <- mkReg (STATE_PIPE);
```

```
25
26    // ================================================================
27    // BEHAVIOR
28
29    ... to be discussed shortly ...
30
31    // ================================================================
32    // INTERFACE
33
34    method Action init (Initial_Params initial_params);
35       ...
36    endmethod
37
38    // Forward in
39    interface fi_RR_to_Retire      = to_FIFOF_I (f_RR_to_Retire);
40    interface fi_Control_to_Retire = to_FIFOF_I (f_Control_to_Retire);
41    interface fi_IALU_to_Retire    = to_FIFOF_I (f_IALU_to_Retire);
42
43    // Speculative DMem
44    interface fi_DMem_S_rsp        = to_FIFOF_I (f_DMem_S_rsp);
45    interface fo_DMem_S_commit     = to_FIFOF_O (f_DMem_S_commit);
46
47    // Non-speculative DMem
48    interface fo_DMem_req       = to_FIFOF_O (f_DMem_req);
49    interface fi_DMem_rsp       = to_FIFOF_I (f_DMem_rsp);
50
51    // Backward out
52    interface fo_F_from_Retire  = to_FIFOF_O (f_F_from_Retire);
53    interface fo_RW_from_Retire = to_FIFOF_O (f_RW_from_Retire);
54 endmodule
```

The first section (preceding BEHAVIOR) instantiates register `rg_epoch` to keep track of the epoch, and then FIFOs for all the incoming and outgoing channels. Finally, it instantiates register `rg_state` to hold the current module state.

The final, INTERFACE, section, as before, has an `init` method, and then lifts the various FIFO interfaces into sub-interfaces for this module.

The BEHAVIOR section consists of a collection of rules, each handling a distinct scenario. In overview:

- Rule `rl_Retire_wrong_path` handles all mis-predicted instructions.
- Rules `rl_Retire_normal` and `rl_Retire_exc` handle instructions with `EXEC_TAG_RETIRE`, *i.e.,* instructions that only have a direct message from the Register-Read-and-Dispatch stage, with nothing in any of the other execution pipelines.
- Rules `rl_Retire_Control_normal` and `rl_Retire_Control_exc` handle instructions with `EXEC_TAG_CONTROL`, *i.e.,* BRANCH and JAL/JALR instructions that came through the Execute Control pipe.
- Rules `rl_Retire_Int_normal` and `rl_Retire_Int_exc` handle instructions with `EXEC_TAG_INT`—LUI, AUIPC and Integer Arithmetic/Logic instructions that came through the Execute Int pipe.

- Rules `rl_Retire_DMem_S_normal` and `rl_Retire_DMem_S_exc` handle instructions with `EXEC_TAG_DMEM`—LOAD and STORE instructions that came through the Execute DMem pipe—and were executed speculatively (into memory-like addresses).
- Rules `rl_Retire_DMem_deferred` and `rl_Retire_DMem_rsp` handle instructions with `EXEC_TAG_DMEM`—LOAD and STORE instructions that came through the Execute DMem pipe—and were deferred (not executed) because they were for MMIO/non-memory-like addresses.

Each pair of rules `rl_XXX_normal` and `rl_XXX_exc` handle the two cases where the instruction completes normally *vs.* the instruction has raised an exception (trap).

### 13.9.1 Common facilities used by many rules

The following function captures the actions to be taken when we discover that the prediction for the successor to this instruction was wrong. The boolean `mispredicted` indicates whether the prediction was correct or not. If the prediction was correct, no action is taken. Otherwise, we increment the epoch, and send a redirection message to the Fetch stage with the correct PC and the new epoch. The disposition of this message was discussed in Section 13.3.

(In file:src_Fife/S5_Retire.bsv)

```
1  // Redirect Fetch stage on mispredicted PC
2  function Action fa_redirect_Fetch (Bool         mispredicted,
3                                     RR_to_Retire x1,
4                                     Bit #(XLEN)  next_pc);
5     action
6        if (mispredicted) begin
7           let next_epoch = rg_epoch + 1;
8           rg_epoch <= next_epoch;
9           let y = F_from_Retire {inum:       x1.inum,
10                                 pc:         x1.pc,
11                                 instr:      x1.instr,
12                                 next_pc:    next_pc,
13                                 next_epoch: next_epoch};
14          f_F_from_Retire.enq (y);
15       end
16    endaction
17 endfunction
```

The following function captures the actions to be taken for updating an instruction's destination rd register. It simply assembles a `RW_to_Retire` message and sends it to the `RR_RW` module. The disposition of this message was discussed in Section 13.5.2.

(In file:src_Fife/S5_Retire.bsv)

```
1  // Update Rd for those instructions that have an Rd
2  function Action fa_update_rd (RR_to_Retire x1, Bool commit, Bit #(XLEN) rd_val);
3     action
4        let y = RW_from_Retire {inum:       x1.inum,
5                                pc:         x1.pc,
6                                instr:      x1.instr,
7                                rd:         instr_rd (x1.instr),
8                                commit:     commit,
9                                data:       rd_val};
10       f_RW_from_Retire.enq (y);
11    endaction
12 endfunction
```

The following definitions examine the first element of the `f_RR_to_Retire` (direct path) queue, which controls how incoming instructions are merged and disposed of, in the rules that follow.

(In file:src_Fife/S5_Retire.bsv)

```
RR_to_Retire x_rr_to_retire = f_RR_to_Retire.first;

Bool wrong_path = (x_rr_to_retire.epoch != rg_epoch);
Bool is_Direct  = (x_rr_to_retire.exec_tag == EXEC_TAG_RETIRE);
Bool is_Control = (x_rr_to_retire.exec_tag == EXEC_TAG_CONTROL);
Bool is_Int     = (x_rr_to_retire.exec_tag == EXEC_TAG_IALU);
Bool is_DMem    = (x_rr_to_retire.exec_tag == EXEC_TAG_DMEM);
```

The incoming instruction is a wrong-path instruction if its accompanying epoch does not match our `rg_epoch` register. The remaining four definitions summarize the class of the instruction based on the execution tag; these control which of the following rules will fire.

### 13.9.2 Rule to discard wrong-path instructions

For a wrong-path instruction, we must dequeue it from `f_RR_to_Retire` and any associated Execute queue (Control, Int or DMem). It if is a DMem STORE instruction that was performed speculatively, we must also send a "discard" message to the head of the store-buffer. Finally, if the instruction has an rd destination register, we must send an discard-scoreboard-reservation message to the RR-RW module using the `fa_update_rd` function described in the Section 13.9.1.

(In file:src_Fife/S5_Retire.bsv)

```
rule rl_Retire_wrong_path ((rg_state == STATE_PIPE)
                           && wrong_path);
   f_RR_to_Retire.deq;

   if (is_Control) f_Control_to_Retire.deq;
   if (is_Int)     f_IALU_to_Retire.deq;
   if (is_DMem) begin
      f_DMem_S_rsp.deq;

      // Send discard to Store Buffer, if needed
      Bool commit_store_val = ((! is_LOAD (x_rr_to_retire.instr))
                               && (! is_LR (x_rr_to_retire.instr))
                               && (f_DMem_S_rsp.first.rsp_type == MEM_RSP_OK));
      if (commit_store_val) begin
         let y = Retire_to_DMem_Commit{inum:   x_rr_to_retire.inum,
                                       commit: False};
         f_DMem_S_commit.enq (y);
      end
   end

   // Release rd scoreboard reservation, if has one
   Bool rd_commit = False;
   if (x_rr_to_retire.has_rd) fa_update_rd (x_rr_to_retire, rd_commit, ?);
endrule
```

### 13.9.3 Rules to retire instructions direct from RR-RW

The following rules handle instructions that are direct from the RR-RW stage, *i.e.,* which were *not* also sent through any of the Execute pipes (Control, Int or DMem).

If it is not an exception, then it must be a CSRRx instruction, which we have not yet implemented yet (we will do so in Chapter **??**). For now, we treat it as a no-op. We dequeue the instruction from `f_RR_to_Retire`. For CSRRx instructions, the correct next-PC is the fall-through PC, so we check this against the predicted PC, and redirect if necessary using action-function `fa_redirect_fetch()` which was described in Section 13.9.1.

(In file:src_Fife/S5_Retire.bsv)

```
rule rl_Retire_normal ((rg_state == STATE_PIPE)
                       && (! wrong_path)
                       && is_Direct
                       && (! x_rr_to_retire.exception));
   f_RR_to_Retire.deq;

   $display ("CSRRx instructions not yet implemented; no-op for now");

   // Redirect Fetch PC if mispredicted
   Bool mispredicted = (x_rr_to_retire.predicted_pc
                        != x_rr_to_retire.fallthru_pc);
   fa_redirect_Fetch (mispredicted, x_rr_to_retire, x_rr_to_retire.fallthru_pc);
endrule
```

If the first element of `f_RR_to_Retire` is carrying an exception, this could be due to a memory-fault during Fetch, or detection of an illegal instruction during Decode. In Chapter **??** we will describe how to handle exceptions, but for now we use `$finish()` to abort the simulation.

(In file:src_Fife/S5_Retire.bsv)

```
rule rl_Retire_exc ((rg_state == STATE_PIPE)
                    && (! wrong_path)
                    && is_Direct
                    && x_rr_to_retire.exception);
   f_RR_to_Retire.deq;

   $display ("Exception-handling not yet implemented");
   $finish (1);
endrule
```

### 13.9.4 Rules to retire instructions from the Execute Control path

If the instruction is a Control instruction without an exception, we pop the relevant incoming queues (`f_RR_to_Retire` and `f_Control_to_Retire`), update the Rd destination register (JAL and JALR often save a "return address" in rd), and redirect the Fetch stage if mispredicted.

(In file:src_Fife/S5_Retire.bsv)

```
rule rl_Retire_Control_normal ((rg_state == STATE_PIPE)
                               && (! wrong_path)
                               && is_Control
                               && (! f_Control_to_Retire.first.exception));
   f_RR_to_Retire.deq;
   let x2 <- pop_o (to_FIFOF_O (f_Control_to_Retire));

   // Updata rd if valid
   Bool rd_commit = True;
   if (x2.data_valid) fa_update_rd (x_rr_to_retire, rd_commit, x2.data);

```

```
12      // Redirect Fetch PC if mispredicted
13      Bool mispredicted = (x_rr_to_retire.predicted_pc != x2.next_pc);
14      fa_redirect_Fetch (mispredicted,  x_rr_to_retire, x2.next_pc);
15   endrule
```

If the instruction is carrying an exception this is because the BRANCH, JAL or JALR target was misaligned. In Chapter **??** we will describe how to handle exceptions, but for now we use `$finish()` to abort the simulation.

(In file:src_Fife/S5_Retire.bsv)

```
1    rule rl_Retire_Control_exc ((rg_state == STATE_PIPE)
2                                && (! wrong_path)
3                                && is_Control
4                                && f_Control_to_Retire.first.exception);
5       f_RR_to_Retire.deq;
6       let x2 <- pop_o (to_FIFOF_O (f_Control_to_Retire));
7
8       $display ("Exception-handling not yet implemented");
9       $finish (1);
10   endrule
```

### 13.9.5 Rules to retire instructions from the Execute Int path

The two rules to retire instructions from the Execute Int path are similar to the rules for the Control path in the previous section.

If the instruction is without an exception, we pop the relevant incoming queues (`f_RR_to_Retire` and `f_IALU_to_Retire`), update the Rd destination register, and redirect the Fetch stage if mispredicted.

(In file:src_Fife/S5_Retire.bsv)

```
1    rule rl_Retire_Int_normal ((rg_state == STATE_PIPE)
2                               && (! wrong_path)
3                               && is_Int
4                               && (! f_IALU_to_Retire.first.exception));
5       f_RR_to_Retire.deq;
6       EX_to_Retire x2 <- pop_o (to_FIFOF_O (f_IALU_to_Retire));
7
8       // Updata rd if valid
9       Bool rd_commit = True;
10      if (x2.data_valid) fa_update_rd (x_rr_to_retire, rd_commit, x2.data);
11
12      // Redirect Fetch PC if mispredicted
13      Bool mispredicted = (x_rr_to_retire.predicted_pc != x_rr_to_retire.fallthru_pc);
14      fa_redirect_Fetch (mispredicted, x_rr_to_retire, x_rr_to_retire.fallthru_pc);
15   endrule
```

If the instruction is carrying an exception: in Chapter **??** we will describe how to handle exceptions, but for now we use `$finish()` to abort the simulation.

(In file:src_Fife/S5_Retire.bsv)

```
1    rule rl_Retire_Control_exc ((rg_state == STATE_PIPE)
2                                && (! wrong_path)
3                                && is_Control
4                                && f_Control_to_Retire.first.exception);
```

```
5       f_RR_to_Retire.deq;
6       let x2 <- pop_o (to_FIFOF_O (f_Control_to_Retire));
7
8       $display ("Exception-handling not yet implemented");
9       $finish (1);
10   endrule
```

Note, none of the standard RISC-V Integer instructions raise any exceptions, so we do not expect this rule to ever be executed. However, if we extend the ISA to new Integer instructions that could raise an exception, this rule is ready to field those exceptions.

### 13.9.6 Rules to retire instructions from the Execute DMem path

From the Execute DMem path we receive memory responses. If the response type is `MEM_RSP_OK` then this instruction was executed speculatively and successfully; we just have to retire it just like the Control and Int scenarios above, with the additional need to send a "commit" message to the store-buffer if it wrote to memory.

(In file:src_Fife/S5_Retire.bsv)

```
1    rule rl_Retire_Dmem_S_normal ((rg_state == STATE_PIPE)
2                                  && (! wrong_path)
3                                  && is_DMem
4                                  && (f_DMem_S_rsp.first.rsp_type == MEM_RSP_OK));
5       f_RR_to_Retire.deq;
6       let x2 <- pop_o (to_FIFOF_O (f_DMem_S_rsp));
7
8       Bool rd_commit = True;
9       if (x_rr_to_retire.has_rd)
10         fa_update_rd (x_rr_to_retire, rd_commit, truncate (x2.data));
11
12      // Send commit to Store Buffer if it wrote to memory
13      Bool wrote_mem = ((! is_LOAD (x_rr_to_retire.instr))
14                        && (! is_LR (x_rr_to_retire.instr)));
15      if (wrote_mem) begin
16         let y = Retire_to_DMem_Commit{inum:   x_rr_to_retire.inum,
17                                       commit: True};
18         f_DMem_S_commit.enq (y);
19      end
20
21      // Redirect Fetch PC if mispredicted
22      Bool mispredicted = (x_rr_to_retire.predicted_pc != x_rr_to_retire.fallthru_pc);
23      fa_redirect_Fetch (mispredicted, x_rr_to_retire, x_rr_to_retire.fallthru_pc);
24   endrule
```

If the DMem response indicates that it attempted a speculative access and encountered an exception, the memory response type will be `MEM_RSP_ERR` or `MEM_RSP_MISALIGNED`. We compute the appropriate RISC-V exception `cause` code. In Chapter **??** we will describe how to handle exceptions, but for now we use `$finish()` to abort the simulation.

(In file:src_Fife/S5_Retire.bsv)

```
1    Bool dmem_S_rsp_exception = ((f_DMem_S_rsp.first.rsp_type    == MEM_RSP_ERR)
2                                 || (f_DMem_S_rsp.first.rsp_type == MEM_RSP_MISALIGNED));
3
```

```
4     rule rl_Retire_Dmem_S_exc ((rg_state == STATE_PIPE)
5                                && (! wrong_path)
6                                && is_DMem
7                                && dmem_S_rsp_exception);
8        f_RR_to_Retire.deq;
9        let x2 <- pop_o (to_FIFOF_O (f_DMem_S_rsp));
10
11       Bit #(XLEN)  cause = ?;
12       if (x2.rsp_type == MEM_RSP_MISALIGNED)
13          cause = (is_LOAD (x_rr_to_retire.instr)
14                   ? cause_LOAD_ADDRESS_MISALIGNED
15                   : cause_STORE_AMO_ADDRESS_MISALIGNED);
16       else
17          cause = (is_LOAD (x_rr_to_retire.instr)
18                   ? cause_LOAD_ACCESS_FAULT
19                   : cause_STORE_AMO_ACCESS_FAULT);
20
21       $display ("Exception-handling not yet implemented");
22       $finish (1);
23    endrule
```

If the DMem response indicates that it deferred the request because the address was to an MMIO/non-memory-like address, the memory response type will be `MEM_RSP_ERR` or `MEM_REQ_DEFERRED`. In this case, we now construct the memory response and send it to memory. Finally we change the module state to `STATE_DMEM_RSP` indicating that we will now await the memory response. Note, this all previous rules will no longer fire because they all have "`(rg_state == STATE_PIPE)`" in their rule conditions, which is now false.

(In file:src_Fife/S5_Retire.bsv)

```
1     rule rl_Retire_DMem_deferred ((rg_state == STATE_PIPE)
2                                   && (! wrong_path)
3                                   && is_DMem
4                                   && (f_DMem_S_rsp.first.rsp_type == MEM_REQ_DEFERRED));
5        let x2 <- pop_o (to_FIFOF_O (f_DMem_S_rsp));
6        let mem_req = Mem_Req{inum:     x2.inum,
7                              pc:       x2.pc,
8                              instr:    x2.instr,
9                              req_type: x2.req_type,
10                             size:     x2.size,
11                             addr:     x2.addr,
12                             data:     x2.data};
13       f_DMem_req.enq (mem_req);
14       rg_state <= STATE_DMEM_RSP;
15    endrule
```

The final rule, handles the corresponding responses from memory. We pop the direct-path information (`f_RR_to_Retire`) and the memory-response (`f_DMem_Rsp`).

If the response returned an exception, we compute the RISC-V exception `cause`. In Chapter **??** we will describe how to handle exceptions, but for now we use `$finish()` to abort the simulation.

Next we check for `MEM_REQ_DEFERRED` responses. This is a redundant check in that it should be impossible—the non-speculative DMem port should never defer any requests—and is just a bit of defensive programming in case of a bug in the memory system. We abort the simulation with `$finish()`.

On successful responses (`MEM_RSP_OK`) we perform the usual writeback-register and possible redirection on misprediction.

Finally, we set `rg_state` to `STATE_PIPE`, which once again enables all the pipeline rules.

(In file:src_Fife/S5_Retire.bsv)

```
rule rl_Retire_DMem_rsp (rg_state == STATE_DMEM_RSP);
   f_RR_to_Retire.deq;
   let mem_rsp <- pop_o (to_FIFOF_O (f_DMem_rsp));

   Bool exception = ((mem_rsp.rsp_type == MEM_RSP_ERR)
                     || (mem_rsp.rsp_type == MEM_RSP_MISALIGNED));
   Bit #(XLEN)  cause = ?;
   if (exception) begin
      if (mem_rsp.rsp_type == MEM_RSP_MISALIGNED)
         cause = (is_LOAD (x_rr_to_retire.instr)
                  ? cause_LOAD_ADDRESS_MISALIGNED
                  : cause_STORE_AMO_ADDRESS_MISALIGNED);
      else
         cause = (is_LOAD (x_rr_to_retire.instr)
                  ? cause_LOAD_ACCESS_FAULT
                  : cause_STORE_AMO_ACCESS_FAULT);
      $finish (1);
   end
   else if (mem_rsp.rsp_type == MEM_REQ_DEFERRED) begin
      $display ("IMPOSSIBLE: Unexpected MEM_REQ_DEFERRED; FINISH.");
      $finish (1);
   end
   else begin
            // mem_rsp.rsp_type will be MEM_REQ_OK here

      // Writeback register
      Bool rd_commit = True;
      if (x_rr_to_retire.has_rd)
         fa_update_rd (x_rr_to_retire, rd_commit, truncate (mem_rsp.data));

      // Redirect Fetch to correct mispredicted PC
      fa_redirect_Fetch ((x_rr_to_retire.predicted_pc != x_rr_to_retire.fallthru_pc),
                         x_rr_to_retire,
                         x_rr_to_retire.fallthru_pc);
   end
   // Go back to pipeline behavior
   rg_state <= STATE_PIPE;
endrule
```

Note that in this module, when we have a DMem response with response-type `MEM_REQ_DEFERRED` we effectively switch from pipeline processing to Drum-like FSM processing (one FSM state issues a DMem request and another FSM state fields the response and processes it). This is a standard trick: for an instruction that cannot for some reason be pipelined, we execute it completely in the Retire module in Drum-like FSM mode. In fact, we shall use exactly this trick to implement CSRRxx instructions (in Chapter **??**, because CSRRx instructions are tricky to execute in the main pipeline.

# Chapter 14

# BSV: Rules, Clocks and Rule Scheduling


**14.1 Rules, Actions and clocks**

**14.2 Rule semantics**

**14.3 Rules scheduling. Example: Counter with `incr` and `decr` methods**

**14.4 CRegs**

**14.5 Implementing mkPipelineFIFOF and mkBypassFIFOF with CRegs**

**14.6 Saving a cycle for Redirect and RW/Scoreboard-write**

# Chapter 15

# BSV: Suggested further study


**15.1 Alternative simulators: Bluesim, other Verilog sims**

**15.2 Static elaboration vs. execution**

**15.3 First-class modules**

**15.4 Polymorphism**

**15.5 Typeclasses. Conversion of Integer literals. Bits/pack/unpack**

**15.6 Bluecheck**

**15.7 Tagged unians, pattern-matching**

**15.8 BH alternative syntax**

# Chapter 16

# RISC-V: Suggested further study


## 16.1 Advanced branch prediction

Is a form of online machine-learning (past history is the "training data").

Branch instruction taken/not-taken hints.

Hysteresis in prediction.

Branch-Target buffers (BTBs)

Return-Address Stacks (RASs)

## 16.2 Memory systems: TCMs, Caches, PMPs

Separate I- and D-caches.

FENCE.I: for "manual" I- and D-Mem coherence.

FENCE: flushing caches for for devices.

Multi-level caches and cache hierarchies.

Non-blocking caches.

Cache-coherence.

Memory Protection with PMPs.

## 16.3 M Extension

Integer Multiply and Divide

## 16.4 A extention

AMO operations

LR, SC, AMOxxx instructions and their implementation in the memory system.

## 16.5 F and D Extensions

IEEE single- and double-precision floating point.

## 16.6 C Extension

Compressed instructions

Implementation: wholly in Decode stage. Can be shared between Fife and Drum.

Affect of compressed instructions on Fetch.

Affect of compressed instructions on PC-prediction.

Affect of non-32-bit alignment of compressed instructions.

## 16.7 Virtual Memory

Page Tables, TLBs, Virtual Memory.

## 16.8 Performance measurement

CSRs TIME, MCYCLE.

Other "hpmcounter" CSRs for other events. Counter enables.

## 16.9 Testing

ISA tests.

Tandem Verification

Sail formal model.

ACTs (sp ?)

## 16.10 Interrupts

- General concepts: CSRs MIP and MIE; minimal MSTATUS with interrupt-enable bits
- Interrupts are initially disabled using the MSTATUS.interrupt-enable bit immediately; CSRxx can be used to re-enable.
- MMIO addresses MTIME, MTIMECMP.
- Interrupts are handled just like traps; the only question is: when to check for interrupts and respond.

- How does MIE bit return to 0?

PLIC, CLIC

Interrupt/trap delegation.

## 16.11 Linux and server-class capability

Multiple privilege levels: Machine, Supervisor, User

### 16.11.1 Hypervisor support

### 16.11.2 RISC-V ISA Formal Specification

Sail model

# Appendix A

# Resources: Documents and Tools

This appendix describes all the resources relevant to this course.

## A.1 GitHub

We will be using GitHub extensively. Course materials will be provided in a public GitHub repository, and GitHub's "discussion" facilities can be used to for questions and answers, visible to all.

For students who do not already know how to use GitHub, we will teach the basics.

More detailed documentation can be found starting at: https://docs.github.com/en/get-started/quickstart

## A.2 RISC-V ISA (Instruction Set Architecture) Specifications

We will refer to the Unpriviledged ISA very frequently, so you may wish to download a copy of the PDF for your laptop, and/or print a copy. The Privileged ISA document is not needed until later.

- "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA".
  Bibliography entry [15] contains a link to `riscv.org` from which to download a PDF.
- "The RISC-V Instruction Set Manual Volume II: Privileged Architecture"
  Bibliography entry [16] contains a link to `riscv.org` from which to download a PDF.

The *formal specification* of the RISC-V ISA is written in the Sail formal-specification language, and can be found at https://github.com/riscv/sail-riscv.


- "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA".
  Bibliography entry [15] contains a link to `riscv.org` from which to download a PDF.
- "The RISC-V Instruction Set Manual Volume II: Privileged Architecture"
  Bibliography entry [16] contains a link to `riscv.org` from which to download a PDF.

## A.3 RISC-V Assembly Language Manuals

We will not do very much assembly language programming, and we will teach whatever notation we need during the course.

There are several RISC-V Assembly Language manuals available online, and some in bookstores; download them only if you prefer a local copy:

- "RISC-V Assembly Programmer's Manual", Palmer Dabbelt, Michael Clark and Alex Bradbury.

  Bibliography entry [3] contains a link to online manual.
- "RISC-V ASSEMBLY LANGUAGE Programmer Manual Part I", Shakti RISC-V Team, Indian Institute of Technology, Madras, India. Please see bibliography entry [13] for link from which to download a PDF.
- "An Introduction to Assembly Programming with RISC-V", Edson Borin.

  Bibliography entry [2] contains a link from which to download a PDF.
- "RISC-V Assembly Language", Anthony J. Dos Reis.

  Bibliography entry [5]. Available in bookstores.

## A.4 RISC-V GNU tools, including `riscv-gcc` compiler

We will be using the GNU tool chain, specifically the *gcc* compiler and linker, and the *objdump* tool for disassembling an ELF file.

During the course we will show you how to install and use these tools.

The use of these tools is mostly the same as when targeting any target architecture, including well-known architectures like x86 and ARM; the student can find voluminous tutorial materials available on the GNU tool chain on web and in books.

*gcc* has some specific options for RISC-V; these are documented here:

- https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html
- https://gcc.gnu.org/onlinedocs/gcc/gcc-command-options/machine-dependent-options/risc-v-options.html

It is also useful to know how to use the GNU debugger tool, *gdb*. Again, the student can find voluminous tutorial materials available on on web and in books.

## A.5 BSV

In this course, we design the hardware of our RISC-V pipelined CPU using the High Level Hardware Description Language **BSV**. The reasons for our choice (instead of using Verilog, SystemVerilog or VHDL) are discussed in more detail in Appendix B of this document, as well as in the Introduction of the "BSV by Example" book described below.

No advance knowledge of **BSV** is needed for this course; we will teach all necessary **BSV** concepts during the course as we go along.

However, for those who would like to study **BSV** on their own, or wish to view additional **BSV** materials, the following sections provide some resources.

### A.5.1 "BSV By Example" book (free downloadable PDF)

This book takes the student through a series of small, targeted **BSV**: examples:

*BSV by Example*, by Rishiyur S. Nikhil and Kathy R. Czeck, 2010.

Quoting from the Introduction:

> "This book is intended to be a gentle introduction to BSV."
>
> " This book tries to take you into the BSV language one small step at a time. Each section includes a complete, executable (and synthesizable) BSV program, and tries to focus on just one feature of the language"

A bound copy of the book can be purchased on Amazon, but a PDF copy of the book and a tar file containing all the BSV program examples in the book can be downloaded for free from the GitHub BSVLang repository at:

https://github.com/BSVLang/Main/tree/master/Tutorials/BSV_Training

- Book (PDF):
  *repository*/`Tutorials/BSV_by_Example_Book/bsv_by_example.pdf`
- Machine-readable version of all examples in the book:
  *repository*/`Tutorials/BSV_by_Example_Book/bsv_by_example_appendix.tar.gz`

### A.5.2 BSV Tutorial

A **BSV** self-paced tutorial is available in the GitHub BSVLang repository:

https://github.com/BSVLang/Main/tree/master/Tutorials/BSV_Training

in the directory *repository*/`Tutorials/BSV_Training/` which looks like this:

```
BSV_Training/
    Build/
    Example_Programs/
        Common
        Eg02a_HelloWorld
        ...
        Eg03a_Bubblesort
        ...
        Eg04a_MicroArchs
        ...
        Eg05a_CRegs_Greater_Concurrency
        ...
        Eg06a_Mergesort
        ...
        Eg09a_AXI4_Stream
    Reference
```

Each of the `Eg*` directories cotains a complete example, along with documentation explaining the example, and instructions on how to compile and Verilog-simulate it. The `Reference` directory contains a collection of lecture slide decks explaining the **BSV** language.

### A.5.3 MIT Course Material

Massachusetts Institute of Technology (MIT) periodically teaches courses on using **BSV** for digital hardware design. The following link:

http://csg.csail.mit.edu/6.375/6_375_2013_www/handouts.html

contains downloadable material:

- PDFs of slide decks for 12 lectures
- PDFs of slide decks for 4 tutorials classes
- PDFs and codes for 6 laboratories

### A.5.4 University of Cambridge Examples

Prof. Simon Moore of University of Cambridge, UK, uses **BSV** in his teaching and research. Several of his **BSV** examples can be found here:

https://www.cl.cam.ac.uk/~swm11/examples/bluespec/

These examples are somewhat more advanced than the ones in the previous sections.

### A.5.5 *bsc* download and installation; *bsc* and BSV manuals

*bsc* is free and open-source, and can be downloaded and installed as described in **BSV**'s GitHub web site https://github.com/B-Lang-org/bsc.

On the main page of that repository you will find links to the following documents (same links also given here):

- The "**BSV** Language Reference Guide". This document describes the syntax and semantics of **BSV**.

  PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/BSV_lang_ref_guide.pdf

- The "BSC Libraries Reference Guide". This document describes the extensive set of libraries and IP (Intellectual Property blocks) available to the **BSV** user.

  PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_libraries_ref_guide.pdf

- The "BSC User Guide". This document describes how to use the *bsc* compiler, which compiles our hardware descriptions written in **BSV** into Verilog (which can then be simulated or synthesizes using standard Verilog tools).

  PDF: https://github.com/B-Lang-org/bsc/releases/latest/download/bsc_user_guide.pdf

We will be using the Language Reference Guide and Librares Reference Guide extensively, so you may wish to download a copy for your laptop.

## A.6 Verilator (or other Verilog simulator)

We will be doing Verilog simulations extensively during this course. For low cost (free), and uniformity, we will be using Verilator.

During the course, we will show you how to install Verilator and use it.

The Verilator web site, https://www.veripool.org/verilator/, contains instructions on how to install Verilator, and also links to PDF and HTML manuals for Verilator. Version 5.004, or any more recent version, will be suitable.

You can use other Verilog simulators if you prefer, but you should independently know how to use them because we cannot offer support during the course. Some possibilities:

- Icarus Verilog, also known as "iverilog". This is a very good, free and open-source, easy-to-use Verilog simulator, but is quite slow compared to other Verilog simulators and so may be less useful for large designs.

  https://steveicarus.github.io/iverilog/usage/getting_started.html

- Commercial simulators from Synopsys, Cadence or Siemens/Mentor Graphics), Aldec, and others. Each of these needs a paid license.

## A.7 Amazon AWS

All hands-on work in this course will be run on the Amazon AWS cloud. This way, everyone in the course has a common, stable, predictable environment and we do not have to waste any time dealing with the countless variations in environments found on different laptops and servers.

During the course, we will explain all necessary concepts as we go along, including how to set them AWS instances and use them.

The Amazon AWS cloud offers, on the "AWS Marketplace" a vast variety of choices for virtual machines or, to use AWS terminology, *instances*. We expect to use the following kinds of instances:

A: An instance running the latest version of Ubuntu (Linux).
B: A so-called "F1 instance", also running Ubuntu. F1 instances have attached FPGAs.
C: An instance running the so-called "AWS FPGA Developer AMI" available in the AWS Marketplace. This runs CentOS (Linux) and comes pre-installed with Xilinx Vivado tools, which we will use for creating FPGA bitfile images during the course.

In Amazon's pricing, (B) is the most expensive, and so we will use that only when we actually run on FPGA. For general development and simulation activities, we'll use (A) which is much cheaper. We will use (C) whenever we're creating a new FPGA bitfile image.

The standard Amazon documentation is can be found here:

- "Set up to use Amazon EC2"
  https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html
- "Tutorial: Get started with Amazon EC2 Linux instances"
  https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html

## A.8 Xilinx Vivado

The FPGAs on Amazon AWS F1 instances are Xilinx Ultrascale FPGAs. Thus, when we build bitfiles on Amazon AWS, we will be using Xilinx Vivado tools (which are provided by AWS for zero incremental cost on AWS FPGA Developer AMI instances, see A.7).

During this course, we will explain all necessary concepts as we go along.

When building a bitfile, it is particularly useful to understand how to interpret the Vivado timing and resource reports. The timing report indicates:

- whether or not our design has successfully met our desired frequency target (MHz), and
- if it did not, which part of our circuit is the likely culprit, which needs to be fixed.

The resource report indicates the "size" or our design (how may LUTs, flip-flops, BRAMs, DSPs, *etc.*).

For more details, Xilinx has extensive documentation for which a good starting point is the "Vivado Design Suite Overview" at

https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/Vivado-Design-Suite-Overview.

## A.9 RISC-V textbooks

This course is self-contained, and it is not necessary to acquire any textbooks.

The following list is provide only as a courtesy and convenience. All these books are written using the RISC-V instruction set as examples, and are available in bookstores.

- "The RISC-V Reader: An Open Architecture Atlas", David Patterson and Andrew Waterman, Strawberry Canyon, 2017. Available in bookstores.

  Bibliography entry [4].

- "Computer Organization and Design RISC-V Edition (2nd Edition): The Hardware Software Interface", David A. Patterson and John L. Hennessy Morgan Kaufman, 2020. Available in bookstores.

  Bibliography entry [11].

- "Computer Architecture: A Quantitative Approach, 6th Edition", John L. Hennessy and David A. Patterson, Morgan Kaufmann, 2017. Available in bookstores.

  Bibliography entry [6]. This is the "classic" textbook on computer architecture, a more advanced textbook.

# Appendix B

# Why BSV?

The BSV language is a modern, high-level, hardware description language with a strong formal semantic basis. It is *fully synthesizable*, *i.e.,* the *bsc* compiler can also compile your source code into Verilog [8], which we regard as the "assembly language" of hardware design. That Verilog code can then be further compiled by ASIC synthesis tools such as Synopsys' Design Compiler or FPGA synthesis tools such as Xilinx Vivado or Altera Quartus, for implementation in ASICs and FPGAs, respectively.

BSV is very suitable for describing architectures precisely and succinctly, and has all the conveniences of modern advanced programming languages such as expressive user-defined types, strong type checking, polymorphism, object orientation and even higher order functions during static elaboration.

All computation in BSV is expressed using "Rules". For many people, this takes a little acclimatization because it is *very* different from traditional programming models (such as in C++ or Java) which are based on sequential processes. But over time, it becomes *the* natural way to think about hardware computation, which is based on massive, fine-grained, heterogeneous parallelism. Complex and high-speed hardware designs are full of very subtle issues of concurrency and ordering, and BSV's computational model is one of the best vehicles with which to study and understand this.

Modern hardware systems-on-a-chip (SoCs) have so much hardware on a single chip that it is useful to conceptualize them, analyze them and design them as *distributed systems* rather than as globally synchronous systems (the traditional view), *i.e.,* where architectural components are loosely coupled and communicate with messages, instead of attempting instantaneous access to global state. This is because the delay in communicating a signal across a chip is now comparable to the clock periods of individual modules. Again, BSV's computational model is well suited to this style of design.

A key to architecting complex systems and reusable modules, whether in software or in hardware, is powerful *interfaces*. Module interfaces in BSV are object-oriented (based on methods), polymorphic/generic, and capture certain computational protocols. This facilitates creating highly reusable modules, enables quick experimentation with alternatives structures, and allows designs to be changed gracefully over time as the requirements and specifications evolve.

Architectural models written in BSV are fully executable. They can be simulated in the Bluesim™ simulator; they can be synthesized to Verilog and simulated on a Verilog simulator; and they can be further synthesized to run on FPGAs or be etched into ASIC silicon, as illustrated in Fig. 1.1. Even when the final target is an ASIC, the ability to run on FPGAs enables early architectural exploration, early development of the software that will later run on the ASIC, and much more extensive and early verification of the correctness of the design. Students are also very excited to see their designs actually running on real FPGA hardware.

In this book, we teach the use of BSV for the design of complex hardware modules and systems by going in detail through a series of examples, and exploring basic concepts as needed along the

way, such as combinational circuits, pipelines, data types, modularity and complex concurrency. At every stage the student is encouraged to run the designs at least in simulation, but preferably also on FPGAs.

By the end of the course we will have seen all the source code for a complete simple, pipelined RISC-V CPU along with a small "SoC" (System-on-a-Chip) including an interconnect, a connection to memory, and a few devices such as a UART.

### Who uses BSV?

BSV has been used in teaching and research at major universities, including MIT (Massachusetts Institute of Technology, USA), Universisty of Cambridge (UK), Indian Institute of Technology, Madras (India), Indian Institute of Technology, Mumbai (India), Seoul National University (South Korea), University of Texas at Austin (USA), Carnegie Mellon Univerity (USA), Georgia Institute of Technology (USA), Cornell University (USA) and Technical University of Darmstadt (Germany).

BSV has been used to design major IP components in commercial ASICs from Texas Instruments, ST Microelectronics and Google. It has been used for FPGA-based modeling at IBM, Intel, Qualcomm, Microsoft Research, several DARPA projects, and others. It is being used for commercial RISC-V processors from InCore Semiconductors (Shakti line of RISC-V processors, India) and The C-DAC (Center for Development of Advance Computing) (Vega line of RISC-V processors, India).

## B.1 Why BSV instead of some other Hardware Design Language?

*The rest of this chapter is intended for those interested in comparing BSV's approach to other approaches (Verilog, SystemVerilog, VHDL, and SystemC), and can be safely skipped by others who just want to get on with learning BSV.*

One may be curious why the material in this book could not have been covered using one of the more widely known languages for hardware design: Verilog [8], VHDL [7], SystemVerilog [10], SystemC [9]. There are several reasons, outlined below.

In the following paragraphs, we will refer to all the above languages, or at least their synthesizable subsets, as "RTL" (Register Transfer Level languages).

### B.1.1 A better computational model

Paradoxically, the formal definitions of the semantics of traditional hardware design languages (HDLs)— Verilog, SystemVerilog, VHDL and SystemC— are not in terms of hardware concepts, but in terms of *software simulation* on conventional computers. Like traditional software programming languages, they are defined in terms of sequential statement execution, with traditional conditionals, loops, and procedure calls and returns, reading and writing conventional variables. Programs can have multiple concurrent *processes* (e.g., "always blocks" in Verilog), but each of them is defined with traditional sequential programming semantics.

Digital hardware, on the other hand, has a quite different computation model. It consists of hundreds, if not thousands of concurrent "state machines" that transform the current state of the hardware, implemented using registers, memories and FIFOs. By and large, there is no sequencing of these state machines based on program counters or statement sequences. Rather, these state machines are independent and "reactive", *i.e.,* each one performs an action whenever certain conditions hold, e.g., when a register holds a particular value, or a value is available in a FIFO, etc.

To bridge this rather large gap from conventional sequential processes to concurrent reactive state machines requires a major mental shift. One must severely restrict code to only a much smaller

so-called *synthesizable subset* of a conventional HDL. Processes are restricted to simple clocked loops: "`always @posedge CLK ...`", also known as an "always-block". Even more draconian is a transposition away from the natural concurrent state machine view to a *state element-centric* view: even though a state element may be read and written by multiple state machines, all updates to that state element must be concentrated in a single always-block, usually in a large conditional construct (if-then-else, case, ...) that describes all the different contributions of different state machines. This transposition, from the natural state machine-centric view to the rather unnatural state element-centric view, is necessary because in the the synthesizable subset there is no synchronization between always-blocks; the programmer has to plan every detail of how to resolve (arbitrate) competing updates to each state element.

In other words, in conventional HDLs, neither the simulation view (sequential processes) nor the synthesizable view (state element-centric always blocks) are a natural way to model hardware behavior.

BSV programs, instead, directly express the natural model of hardware— concurrent state machines. Each "rule" in BSV is a reactive state transition that awaits some condition on the hardware state and then takes an action to transforms the state. Further, each rule is an *atomic transaction*, *i.e.,* the details of how one arbitrates competing accesses from multiple rules to common shared state is left to the compiler. This kind of arbitration logic, which is hand-written in other HDLs, is a major source of bugs.

In BSV, unlike in other HDLs, the semantics are identical whether you execute in simulation or in hardware—there is no mental gear shift necessary, and simulation behavior is always identical to synthesized hardware behavior.

Finally, the Rules computation model uniquely encourages *refinement*, a powerful design methodology. We initially create a high-level, approximate model of a target design, using a few large rules. Both the level of micro-architectural detail and the range of functionality are approximated (abstracted). Often such a model can be written in less than a day, and it can immediately be executed to verify functional correctness. Then, over time, we incrementally add architectural detail—for example, pipeline registers and state machines with more, smaller steps—and the original rules (large step state transitions) are replaced by more, smaller rules (small step state transitions). The atomic semantics of rules makes this a robust methodology, *i.e.,*, a refinement does not have a large ripple effect. This is in quite dramatic contrast to the difficulty in changing RTL, which is notoriously brittle and unforgiving.

Refinement allows early and continous confidence in functional correctness and completeness, since we execute the code very frequently. Refinement allows mid-course corrections in functionality, after observing execution on real data. Refinement allows separating *functionality* from *performance*, achieving functionality early and holding it constant while we improve performance to meet a performance target (by target performance we mean some desired targets for speed, area, and power).

### B.1.2 Modern language features

The field of programming languages has seen tremendous progress since the early days (1950s). Modern high-level languages have advanced type systems (polymorphism, typeclasses and overloading, functional types, and so on). Modern high-level languages have strong mechanisms for encapsulation and abstraction (such as object-orientation) which promote the separation of concerns between externally visible behavior and internal representation choices. Modern high-level languages make frequent use of higher-order functions—functions whose arguments and results can themselves be functions and data structures whose components can be functions.

Unfortunately, practically none of these powerful features are present in the synthesizable subsets of conventional HDLs[1]. BSV, on the other hand, adopts the full power of the Haskell functional pro-

[1]SystemVerilog and SystemC have object-orientation, polymorphism, and overloading, but these are

gramming language [12]: algebraic types, functional types, polymorphic types, typeclasses, higher-order functions, and recursive and monadic static elaboration. This delivers unprecedented expressive power, type safety and type flexibility in a hardware design language.

### B.1.3 Comparison with C++-based High Level Synthesis

Recently, some tools have become available under the rubric of "High Level Synthesis" (HLS) that claim to shield you from this mental gear shift from simulation to hardware. Designs are written in a traditional sequential programming language (typically C++), and an HLS tool automatically compiles this into a hardware implementation. While beautiful in concept, there are many serious limitations in practice, which are discussed below.

#### C++ codes need significant rewriting

C++ HLS tools will rarely accept arbitrary, off-the-shelf C++ codes and produce good hardware implementations. C++ codes often requre significant restructuring to achieve good results.

First, the tools only accept a limited subset of C++ syntax. In particular, these tools are very averse to any kind of pointer-based argument passing or data structures, unless all the pointers can be resolved by the compiler (*i.e.,*, the compiler statically knows the addresses represented by the pointers). This is because, while C++ normally executes on machines that provide the abstraction of a single large memory with a single address space (so a pointer is fundamentally an address, and dynamic allocation and relocation are easy), hardware designs typically use hundreds or thousands of individual memory units, from registers to register files to SRAMs, DRAMs, Flash memories, and ROMs, each with its own address space.

Second, most C++ codes written for conventional execution rely deeply on sequential execution. For example, they may re-use a variable (multiple reads and writes in different phases of the code). Many of these programming techniques, often a good idea for higher performance and smaller memory footprints in conventional execution, are exactly the opposite of what is needed for hardware implementation, which is highly parallel.

Overall, for good results, one must develop a keen sense of the hardware implementation impact of various "styles" of writing C++ code. Small changes in style can mean the difference between a terrible implementation and an acceptable one. One vendor insists that any team adopting their tool should not consist solely of C++ experts, but must also include hardware engineers.

#### Narrow range of applicability due to automatic parallelization

C++ is, by official definition, a completely sequential language. Hardware, on the other hand, relies on massive, fine-grain parallelism. It is the HLS tool that has to pull off this magical transformation.

C++ HLS tools rely on a body of knowledge called CDFG Analysis (Control and Data Flow Graph Analysis). After parsing and typechecking, the C++ program is represented internally in a data structure called the CDFG. This CDFG, initially directly reflecting the sequential nature of the source, is analyzed and transformed into a parallel representation from which, eventually, hardware is generated.

It turns out that this transformation only works well for a narrow range of program structures—cleanly nested `for`-loops with fixed iteration bounds, operating on dense rectangular arrays. Of course, many signal-processing and image-processing applications do have this structure, and C++ HLS tools have found their greatest success in this arena.

---

typically used only in simulation for verification environments of hardware designs, not for actual hardware design itself.

But the moment we step outside this sweet spot, towards sparse arrays or programs that are highly control-dominated, these tools fall off a cliff. Most hardware design in fact involves components that don't fall into the C++ HLS sweet spot: CPUs, cache systems, switched interconnects, flash memory and disk controllers, high-speed I/O controllers for Ethernet, PCIe, USB, and so on. For example, we are unaware of any project using C++ HLS for CPU design, whereas there are over a dozen such projects using BSV.

### Lack of "Algotecture": Architectural transparency and predictability

Most people with some training in Computer Science are familiar with the idea that Algorithms are Job One—when writing performance-critical software, the first-order concern is to design a good algorithm. Further, creating a good algorithm is a creative act; compilers don't automatically create good algorithms for you[2].

Unfortunately, because most of our codes run on classical von Neumann machines, many people forget that, when the execution platform changes, our old algorithms may no longer be any good—the assumptions about the cost of fundamental operations may no longer valid and in fact may be wildly different, requiring a complete re-think of the algorithm.

This bring up a fundamental difference between software design and hardware design. In software, you are given a particular target architecture (CPU, GPU, cluster, vector machine, ...), and the designer's job is to design a good algorithm for that fixed architecture. In hardware, on the other hand, the designer's job is to design the algorithm and the architecture *jointly*. In other words, for hardware designers, algorithm and architecture are joined at the hip; it is meaningless to separate these activities. We thus use the term *Algotecture* to describe this integrated activity.

Unfortunately, most C++ HLS tools provide very narrow visibility and control into architecture. For example, directives for loop unrolling and loop fusion may allow you to express some variation in iterative *vs.* parallel *vs.* pipelined structures. But, basically, it's the tool that chooses the architecture, and you have some weak knobs to guide its choices. A common syndrome with C++ HLS tools is that one quickly produces an implementation, but it is terrible in area or performance, and this is followed by a *long* tail of activity in which the designer tweaks the knobs every which way in an effor to beat it down into the desired performance envelope.

In contrast, with BSV, architectural choices (like algorithmic choices) are in the hands of the designer, where it should be. There are no surprises with respect to architecture; performance is never a mystery, and the designer can quickly improve it and converge to an acceptable solution.

### Summary

In summary, it is our experience that BSV is a much better language for complex hardware design, whether control or data oriented, whether for modeling or architectural exploration or final implementation, or for synthesizable on-FPGA verification transactors. Following the philosophy of DSLs (Domain Specific Languages), BSV is very much an expressive DSL beautifully suited for hardware design, whereas sequential C++ is certainly not (it was never intended to be!).

[2] Of course, there is research in this area, but this starts entering the realm of Artificial Intelligence.

# Appendix C

# Glossary

**2's Complement** See entry for "Two's Complement".

**ASIC** Application-Specific Integrated Circuit. A kind of electronic device that represents a desired digital circuit directly in silicon and has been fabricated for that purpose (not customizable and general-purpose like an FPGA).

**API** Application Programming Interface. Term commonly used in many programming languages, methodologies and protocols to describe the set of functions/procedures/methods used to interact with a module/object by external entitities (from outside the module/object). The API clearly separates external concerns from internal concerns. External concerns are about "what" a method does or sequence of methods do: what are their argument and result types, and what do they (abstractly) achieve. Internal concerns are about "how" methods do what they are supposed to do. This separation of concerns also allow transparently substituting a module implementation with an alternate implementation (*e.g.,* for greater efficiency) without disturbing the external context.

**BSV, BH** An open-source, modern, High-Level HDL. Two optional syntaxes (choose to one's taste): BSV has traditional Verilog-like syntax, BH has traditional Haskell-like syntax.

**CPU** Central Processing Unit. The computational element of a computer.

**CSRs** Control and Status Registers. These are special registers in the ISA, most of which are accessibly only while executing at higher privilege levels (Machine and Supervisor). Certain key CSRs play a central role in disciplined transition between privilege levels, in virtual memory, and in memory protection.

**DRAM** Dynamic Random Access Memory. A kind of silicon chip that implements memory. Compared to SRAM, is larger (number of bits), denser (bits per silicon area), cheaper ($ per bit), uses less power (watts per bit) and is more complex to operate (needing regular refreshing *etc.*). Usually off-chip (not part of an ASIC or an FPGA).

**FPGA** Field Programmable Gate Array. A kind of electronic device that has configurable circuits that can be customized to represent any desired digital design. These are catalog parts available from several vendors.

**FPGA Board** A circuit board containing one or more FPGAs, a power supply, and DRAM memories. Often contains other facilities such as GPIO, UARTs, JTAG, PCIe bus connections, Ethernet connection, USB connection, Flash memory, and so on.

**FSM** Finite State Machine. A sequential process that moves ("transitions") from one state to another in a fixed repertoire of states. Transitions may loop back to earlier states, and may conditionally select one of a set of alternative next-states.

FSMs are named after the Greek town of Ephysem, the capital of the Kingdom of Ephyra, whose most famous resident was a gentleman named Sisyphus (https://en.wikipedia.org/wiki/Sisyphus), who repeatedly rolled a boulder up a hill only for it to roll back down again. This was widely reported in a Greek media outlet of the time called the Drudge Report.

**GPIO** General Purpose Input Output. An electronic device attached to a computer system. When the CPU stores a byte/word to a GPIO address, the bits of the word appear as electronic signals from the device, and can be used as an *actuator*—switch on/off a back of LED lamps, a relay, a motor, *etc.*.. When the CPU loads a byte/word from a GPIO address, it can read the state of a *sensor*—switches, photocells, motor speed, temperature, *etc.*.

**GPR** General Purpose Register. For RISC-V, just a synonym for the basic register set holding integers. They are "general purpose" in the sense that software is free to use them in any way (in contrast with some earlier ISAs that restricted certain registers to certain roles, such as holding addresses).

**HDL** Hardware Design Language. A language in which one can represent circuits, and for which there are tools that can render a program into actual circuits for FPGAs and ASICs. Examples include: BSV, BH, Chisel, Verilog, SystemVerilog, VHDL.

**HLHDL** High-Level Hardware Design Language. An HDL with higher-levels of abstraction and more powerful constructs and semantics compared to the traditional HDLs Verilog, SystemVerilog and VHDL, in the same sense that modern software programming languages (Java, Python, Javascript, Haskell, OCaml, ...) have higher-levels of abstraction than C/C++ which, in turn, have higher levels of abstraction than Assembly Language. Examples include BSV, BH (the Haskell-syntax variant of BSV), Chisel, and HLS.

**HLS** High Level Synthesis. The term typically used for tools and methodology that compile C/C++/SystemC programs into hardware. HLS can be fragile in that it works best only on certain subsets of C/C++ ("simple rectangular loop and array" algorithms), and require certain coding styles and directives.

**ISA** Instruction Set Architecture. A specification of instructions: how an instruction is coded in bits; "architectural state" (PC, registers *etc.*); what it means to execute an instruction; assembly language syntax. The specification is described independently of any particular implementation, traditionally in a manual with text and diagrams, occasionally and recently also in a formal-specification language.

An ISA can (and typically does) have many possible implementations, varying widely in speed, size, power, cost, technology (ASIC, FPGA), *etc.* Examples of famous ISAs and vendors who supply implementations include RISC-V (diverse vendors), x86 (Intel and AMD), ARM (Arm, Apple, Samsung, others), Sparc (Sun, Oracle, Fujitsu, others), MIPS (MIPS, Inc.), Power and PowerPC (IBM, others), ...

**Microarchitecture** The structural and behavioral details of an ISA implementation that are *below* the level of abstraction of the ISA, *i.e.,* not demanded by the ISA but chosen by the implementor for practical reasons (speed, power, area, cost, ...). Examples: pipelines, branch prediction, scoreboards, register renaming, out-of-order execution, superscalarity, instruction fission and fusion, replicated execution units, store-buffers, ...

**MMIO** Memory-Mapped Input-Output. In RISC-V, the CPU reads and writes registers in a device using ordinary LOAD and STORE instructions. The memory system interprets the addresses to direct such requests to a device. Using LOADs and STOREs, the CPU can control the device, send data to the device and retrieve data from the device.

**OS** Operating System. Can vary from small, embedded, real-time OSs such as FreeRTOS, to more capable embedded OSs like Zephyr, to secure micro-kernels like seL4, to full-featured OSs like Linux, Windows, MacOS, Solaris, AIX, *etc.*

**RISC-V** A particular standard ISA. Originated circa 2008-2010 in research at University of California, Berkeley, and subsequently spun out (2010s) into an international non-profit consortium "RISC-V International" (RVI) headquartered in Switzerland (https://riscv.org).

Unlike other well-known ISAs, the RISC-V ISA is an *open* standard, *i.e.,* implementors do not need to pay any license fee in order to use the ISA, which is one of the factors behind its wide adoption by hundreds of vendors.

**RTL** Register-Transfer Level/Language. This is a level of abstraction of describing hardware that assumes that the available primitive components are clocked registers and combinational circuits for multiplexers, and basic arithmetic and logic functions (adders, subtractors, boolean operations, shifters, *etc.*).

This is a higher level of abstraction than AND/OR/XOR/NOT gates which, in turn, are a higher level of abstraction than transistors which, in turn, are a higher level of abstraction than silicon regions. Each layer of abstraction is automatically compiled to a lower layer using various tools.

**RVI** RISC-V International. See entry for RISC-V.

**SoC** System-on-a-chip. Refers to a complete computing system on a chip, including one or more CPUs (with MMUs and caches), shared caches, interconnects, DRAM interface, JTAG, accelerators and devices, *etc.*

**SRAM** Static Random Access Memory. A kind of silicon chip that implements memory. See DRAM above for comparison. Usually on-chip in an ASIC or an FPGA.

**SystemVerilog** One of the major HDLs. Originally created in the 2000s as a proper superset of Verilog (and thereby subsuming Verilog), and incorporating many features from VHDL; incorporated some modern features from object-oriented software programming languages (principally used in verification testbenches in simulation only); then an IEEE standard that has gone through several versions. Can be used for both analog and digital circuits. Some features can only be used in simulation (a "synthesizable subset" can be rendered into hardware).

**Two's Complement** A particular representation of positive and negative integers in bits (binary) that makes it possible to perform both addition and subtraction using the same hardware. Wikipedia has a good discussion: https://en.wikipedia.org/wiki/Two%27s_complement

**UART** Universal Asynchronous Receiver/Transmitter. An electronic device attached to a computer system through which the CPU can read ASCII characters from a keyboard and send ASCII characters to a display screen. Typically used for the main console of a computer system.

**Verilog** One of the two grand old HDLs (the other is VHDL). Originally created in the 1980s; then an IEEE standard that has gone through several versions; then subsumed by SystemVerilog. Can be used for both analog and digital circuits. Some features can only be used in simulation (a "synthesizable subset" can be rendered into hardware).

**VHDL** One of the two grand old HDLs (the other is Verilog). Originally created in the 1980s; then an IEEE standard that has gone through several versions. Many features were adopted by SystemVerilog. Can be used for both analog and digital circuits. Some features can only be used in simulation (a "synthesizable subset" can be rendered into hardware).

# Index


`action-endaction` blocks, 9-4
Address alignment, 5-7

BSV
- `/*`, start of block comment (until-`*/`), 4-13
- `//`, start of comment-to-end-of-line, 4-13
- `?` (don't care literal value), 5-4
- `?`, the don't care value, 5-4
- `Bool`, 4-4
- AAAA_AAAA, the default don't care value, 5-4
- `Action` type of expression with side-effects, 4-7
- `Action`: primitive type of actions, 9-3
- `Action`: type of pure side-effect expressions, 7-4
- actions, 9-3
- `ActionValue` type of experssion with side-effects, 4-7
- Binary notation for integer literals, 4-4
- `Bit`
  - Integer type, 4-3
- Bit Vectors, 4-2
  - `extend`, 4-3
  - `truncate`, 4-3
  - `zeroExtend`, 4-3
  - operators on, 4-2
  - slices of, 4-2
- `Bool`
  - operators on, 4-4
- bus (hardware, bundle of wires), 4-6
- Combinational circuits, 4-6
  - data types, 4-8
  - purity, 4-6
- Combinational primitives, 4-6
- Comments
  - block, from `/*` to matching `*/`, 4-13
  - to end-of-line, starting with `//`, 4-13
- Conditional compilation, 4-18
- Connecting FIFOs, 7-10
- deriving
  - Bits, 5-3
  - Fshow, 5-3

`deriving Bits`, 4-12
`deriving Eq`, 4-12
`deriving FShow`, 4-12
`$display` has `Action` type, 9-4
Don't care literal value "?", 5-4
`enum` types, 4-11
field
  of a `struct`, 5-1
FIFO, 7-7
`FIFOF`
  type of stored value, 7-7
`FIFOF` interface, 7-7
`FIFOF` interface methods, 7-7
`FIFO` interface, 7-7
`FIFOF_O`
  interface transformer from `FIFOF`, 7-9
`FIFOF_I` semi-fifo interface, 7-9
`FIFOF_O`: semi-fifo interface, 7-9
Finite State Machines, 9-1
FSMs, 9-1
- concurrent *vs.*. sequential, 9-2
- sequential *vs.*. concurrent, 9-2
- `Stmt`: type of argument to FSM module constructors, 9-5

functions
- application, 4-7
- definition, 4-7

Haskell, monadic types similarity, 4-8
Hexadecimal notation for integer literals, 4-4
Identifer syntax, 4-12
Identifiers
- Enum constant: initial upper-case letter, 4-4, 4-12
- First letter lower- or upper-case, 4-4, 4-12
- Ordinary: initial lower-case letter, 4-4, 4-12
- Type: initial upper-case letter, 4-4, 4-12

if-then-else, 4-13
- nested, 4-14

if-then-else: ordinary expression *vs* `StmtFSM` process, 9-5

`Int`
  Integer type, 4-3
Integer types `Bit`, `Int`, `UInt`, `Integer`, 4-3
Interface
  `FIFOF` FIFO interface, 7-7
  `Reg` register interface, 7-4
  `RegFile` register file interface, 7-6
  type, 7-2
Interface transformer functions, 7-9
internal behavion: rules, 7-2
`let`
  binding an identifier with implicit type declaration, 5-4
Literals
  Binary integer notation, 4-4
  Hexadecimal integer notation, 4-4
member
  of a `struct`, 5-1
Method
  invocation of module method, 7-3
`mkAutoFSM` module in `StmtFSM` library package, 9-6
`mkBypassFIFOF`
  module (constructor), 7-11
`mkConnection` for connecting compatible interfaces, 7-10
`mkFIFOF`
  instantiation, 7-8
  module (constructor), 7-8
  reset value, 7-8
`mkPipelineFIFOF`
  module (constructor), 7-11
`mkReg`
  instantiation, 7-5
  module (constructor), 7-5
  reset value, 7-5
`mkRegU`
  module (constructor), 7-5
`mkSizedFIFOF`
  module (constructor), 7-8
Module, 7-1
  (persisitent) state, 7-1
  behavior, 7-3
  constructor, 7-3
  instance, 7-3
  instantiation, 7-3
  interface, 7-1, 7-3
  method invocation, 7-3
  `mkBypassFIFOF` module (constructor), 7-11
  `mkFIFOF` module (constructor), 7-8
  `mkPipelineFIFOF` module (constructor), 7-11
  `mkReg` module (constructor), 7-5
  `mkRegFileFull` module (constructor), 7-7
  state, 7-3
Monadic types
  Haskell similarity, 4-8
Monomorphic Types (types without type-variables), 7-11
multiplexers, 4-13
  cascaded/serial/priority, 4-13
  parallel, 4-15
MUX, 4-13
Operators on Bit Vectors, 4-2
Overloading: Typeclasses and typeclass instances, 7-11
packing of struct fields and vector elements, 5-3
parameterization, 4-16
Polymorphic Types, 7-11
  Type variables (identifiers beginning with lower-case letter), 7-11
Propagation delay, 4-7
`_read`: register method, 7-4
`RegFile`, 7-6
Register, 7-4
  `_read` method, 7-4
  `Reg` register interface, 7-4
  `_write` method, 7-4
Register file, 7-6
  methods, 7-6
  `RegFile` interface, 7-6
  `RegFile` register file interface, 7-6
  type of index, 7-6
  type of stored value, 7-6
`replicate` `vector` library function to create a vector value, 13-7
`rule`: the fundamental behavioral construct in BSV, 7-11
rules
  internal behavior, internal processes, 7-2
Semi-FIFO
  `FIFOF_I` semi-fifo interface, 7-9
  `FIFOF_O` semi-fifo interface, 7-9
`StmtFSM`, 9-3
  `for`-loop repetition, 9-7
  `mkAutoFSM` module, 9-6
  `await`: pausing until some condition, 9-6
  `if-then-else`: process conditional, 9-5
  `seq .. endseq`: sequences of actions, 9-5
  `while`-loop repetition, 9-6

struct
entire struct values, 5-4
field assignment/update, 5-5
field selection, 5-5
hetoregeneous collection of values, 5-1
Nested, 6-2
`struct`
type declaration, 5-3
structs
packing of fields, 5-3
`(* synthesize *)` attribute, 7-3
`synthesize`
attribute on modules for Verilog generation, 7-3, 7-12
Testbenches, 4-9
FSMs, 4-9
Type variables
Identifiers beginning with lower-case letter, for polymorphism, 7-11
Typeclass
instance of, 4-12, 7-11
Typeclasses, 4-12
BSV's "overloading" mechanism, 7-11
Types
`ActionValue`, 4-7
`Action`, 4-7
`Bit#(n)`, 4-2
`Bool`, 4-4
interface, 7-2
numeric, 4-17
of combinational circuits, 4-8
synonyms, 4-17
valueOf: value of a numeric type, 4-18
`Integer`
Unbounded (non-synthesizable) Integer type, 4-3
`UInt`
Unsigned integer type, 4-3
valueOf: value of a numeric type, 4-18
`vector`
library data type, 13-7
library `replicate` function, 13-7
of `n` bits *vs.* `Bit#(n)`, 13-7
representation in bits, 13-7
vectors
packing of fields, 5-3
Wraparound arithmetic for fixed-width integer types, 4-4
`_write`: register method, 7-4

Decode function (`fn_Decode`), 6-3
Dispatch function (`fn_Dispatch`), 6-7
DMem, Data Memory5-5
Drum
as a set of concurrent FSMs, 9-2
as an FSM, 9-2
CPU interface, 10-1
CPU module behavior, 10-5
Skeleton module, 10-2

Execute Control function (`fn_EX_Control`), 6-11
Execute Integer function (`fn_EX_Int`), 6-13

Fetch function (`fn_Fetch`), 6-1
Fife
as a set of concurrent FSMs, 9-2
Skeleton module, 10-2
FIFO
strongly-typed, 7-8
`fn_Decode` (Decode function), 6-3
`fn_Dispatch` (Dispatch function), 6-7
`fn_EX_Control` (Execute Control function), 6-11
`fn_EX_Int` (Execute Integer function), 6-13
`fn_Fetch` (Fetch function), 6-1

Harvard architecture, 5-6
Self-modifying code, 5-6

IMem, Instruction Memory5-5

JIT (Just-in-time compiling), 5-6
Just-in-time compiling (JIT), 5-6

`let`-bindings in `Action` blocks, 9-4
`let`: BSV, binding an identifier with implicit type declaration, 5-4

Memory
Address alignment, 5-7
Request, 5-6
Response, 5-8
`mkRISCV_GPRs` a module wrapper around library `RegFile`, 8-2
`mkRISCV_GPRs_synth` a module wrapper for `mkRISCV_GPRs` for synthesizability, 8-3

Register
`<=` register assignment, 7-5
implicit register read, 7-5
strongly-typed, 7-4
RISC-V
Architectural state, 3-3
DMem (data memory), 5-5
Drum CPU module behavior, 10-5
Drum skeleton module, 10-2
Fife skeleton module, 10-2

GPRs: general purpose registers, 8-1
Hazards
Scoreboard for managing, 12-4
IMem (instruction memory), 5-5
ISA
Architectural State, 2-2
Assembly Language, 2-2
Contract between software and hardware implementations, 2-2
Enables software portability, 2-2
Evolution of, 2-16
Extensions to, 2-16
Formal Specification in Sail, 2-2
Instruction semantics, 2-2
Instruction Set, 2-2
Instruction Set Architecture, 2-1
Instruction Set encoding in bits, 2-2
PC, Program Counter, 2-2
Program Counter, PC, 2-2
Register File, 2-2
RV32I base integer instructions, 2-7
Sail Formal Specification language, 2-2
Machine code, 2-2
Micro-architecture, 2-2
Multi-core, 2-2
Multi-threaded, 2-2
Out-of-order, 2-2
Pipelining, 2-2
Speculation, 2-2
Superscalar, 2-2
Scoreboard, to manage register read/write hazards, 12-4
`x0`: Special "always zero" register, 8-1
RISCV
Branch prediction, 12-2
Bubble in a pipeline, 12-5
Bypassing, 12-6
Commit the side-effects of an instruction, 12-4
Instruction ordering
tags for, 12-7
Memory access
cache hit/miss, 3-4
latency, 3-4
one or more clock ticks, 3-4
viewed as a pipeline/queue, 3-4
Misprediction (wrong path instructions), 12-2
MMIO (Memory-Mapped Input Output), 12-8
PC prediction, 12-2
redirection, 12-3
Pipeline
Bubble, 12-5
Prediction, 12-2
redirection on misprediction, 12-3
`rg_epoch` register for managing mispredictions, 12-3
Short-circuiting (bypassing), 12-6
Speculation of instructions, 12-4
Speculative instruction, 12-4
Split-phase memory transaction, 3-4
Store Buffer, 12-7
Tags
`EXEC_TAG_CONTROL`, 12-7
`EXEC_TAG_DIRECT`, 12-7
`EXEC_TAG_DMEM`, 12-7
`EXEC_TAG_IALU`, 12-7
for proper instruction ordering, 12-7
Wrong-path due (mispredicted instructions), 12-2
`RISCV_GPRs_IFC` interface for `mkRISCV_GPRs`, 8-1

`truncate`, operation to shrink bit-width, 6-6

# Bibliography


[1] Bluespec, Inc. BSV Guide, 2022 (first version 2000).

[2] E. Borin. *An Introduction to Assembly Programming with RISC-V*. 2021 (Revised: May 9, 2022). PDF online: https://riscv-programming.org/book.html.

[3] P. Dabbelt, M. Clark, and A. Bradbury. RISC-V Assembly Programmer's Manual, Recent update: Jun 29, 2023. Online: https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md.

[4] P. David and W. Andrew. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN-10: 0999249118, ISBN-13: 978-0999249116. Available in bookstores.

[5] A. J. Dos Reis. *RISC-V Assembly Language*. 2019. ISBN-10: 1088462006, ISBN-13: 978-1088462003. Available on Amazon.com.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 6th Edition*. Morgan Kaufmann. The Morgan Kaufmann Series in Computer Architecture and Design. ISBN-10: 0128119055, ISBN-13: 978-0128119051. Available in bookstores.

[7] IEEE. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, 2002.

[8] IEEE. IEEE Standard Verilog (R) Hardware Description Language, 2005. IEEE Std 1364-2005.

[9] IEEE. IEEE Standard for Standard SystemC Language Reference Manual, January 9 2012. IEEE Std 1666-2011.

[10] IEEE. IEEE Standard for System Verilog—Unified Hardware Design, Specification and Verification Language, 21 February 2013. IEEE Std 1800-2012.

[11] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition (2nd Edition): The Hardware Software Interface*. Morgan Kaufman, 2020. The Morgan Kaufmann Series in Computer Architecture and Design. ISBN-10: 0128203315, ISBN-13: 978-0128203316. Available in bookstores.

[12] S. Peyton Jones (Editor). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 5 2003. haskell.org.

[13] SHAKTI Development Team @ iitm '20 shakti.org.in (Indian Institute of Technology, Madras, India). RISC-V ASSEMBLY LANGUAGE, Programmer Manual Part I, 2020. PDF online: https://shakti.org.in/docs/risc-v-asm-manual.pdf.

[14] Various authors. BSV Libraries Reference Guide, 2024 (revised frequently).

[15] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, December 13 2019. Document Version 20191213.. PDF online (and newer versions, if any): https://riscv.org/technical/specifications/.

[16] A. Waterman, K. Asanović, and J. Hauser. The RISC-V Instruction Set Manual Volume II: Privileged Architecture, December 4 2021. Document Version 20211203. PDF online (and newer versions, if any): https://riscv.org/technical/specifications/.