Designing a Simple Pipelined RISC-V CPU
(coding the hardware in BSV)

or

Learning BSV for Digital Design
(using a Simple Pipelined RISC-V CPU as a running example)

Rishiyur S. Nikhil

Bluespec, Inc.

DRAFT: February 9, 2024, 16:30h EDT

Contents

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 CPU 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). Nevertheless, it will be powerful enough to execute Linux, an industrial-strength operating system.

Figure 1.1 shows the plan for topics covered in this book.

- 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 available), but understanding the ISA is of course a prerequisite to informing our design. The RISC-V ISA has many options; our focus will be on a "standard" suite:
  - From the RISC-V Unprivileged ISA spec: basic integer arithmetic and logic operations; branch and jump; load and store; integer multiply and divide; atomic memory operations; floating-point operations; compressed instructions (so-called RV32IMAFDC and RV64IMAFDC).
  - From the RISC-V Privileged ISA spec: handling traps and interrupts; Control and Status Registers (CSRs); Machine, Supervisor and User Privilege levels.
- 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

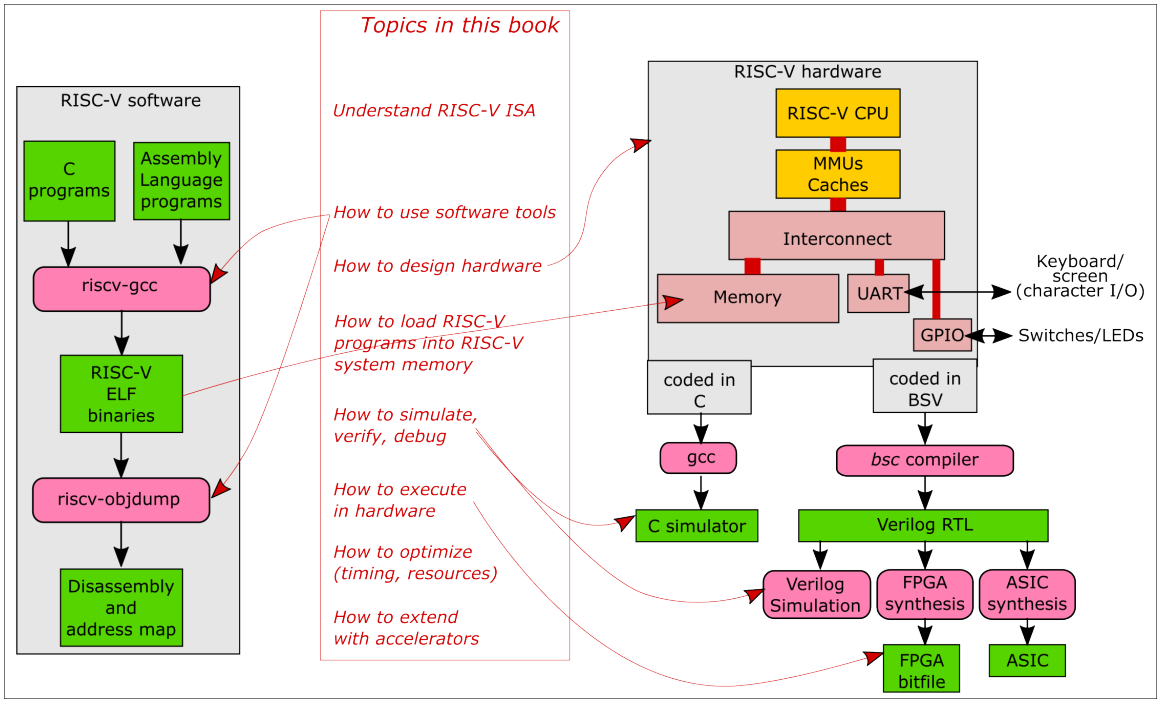

Figure 1.1: Topics covered (in red text in red box)

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.

So, far, all this is not implementation-specific, *i.e.*, it is generic information about RISC-V.

- 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 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. Later we will discuss MMUs (Memory Management Units) and Caches, and possibly other devices and accelerators.

- We will learn how to 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 learn now 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.

We will create two hardware designs. The first design is "Drum", a *non-pipelined* implementation which will familiarize us 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. The second design is "Fife", which has a five-to-six stage, mostly in-order pipeline, which is a microarchitectural change focused on higher performance (speed) than Drum. Both will execute exactly the same binaries; the only difference will be in Fife's superior performance (speed). Both designs will share a large part of the BSV code implementing the essential functionality for executing the RISC-V ISA.

As we work through the two designs, we will concurrently learn how to code in BSV, the 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.

Please see Appendix A for a detailed listing of resources (documents and software tools) needed for this book.

Chapter 2

The RISC-V ISA; and Compiling Programs for the ISA

This chapter contains only generic RISC-V information, not specific to this book. This chapter can be safely skipped by those already familiar with these generic topics, or who choose to learn it elsewhere:

- RISC-V ISA specification documents. Please see Appendix A.2 for links.
- RISC-V Assembly Language manuals Please see Appendix A.3 for links.
- RISC-V Gnu Tools and related documentation. Please see Appendix A.4 for links.
- RISC-V textbooks. Please see Appendix A.9 for links.

Lecture slides exist for this chapter; need to render into narrative text.

NOTE: This chapter will be written later, since there is much information available in the given links.

Chapter 3

Design Space for RISC-V interpreters: 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 9.

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. Drum is covered from Chapter 10 through Chapter 10.

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" [14]. 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"[15].

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.

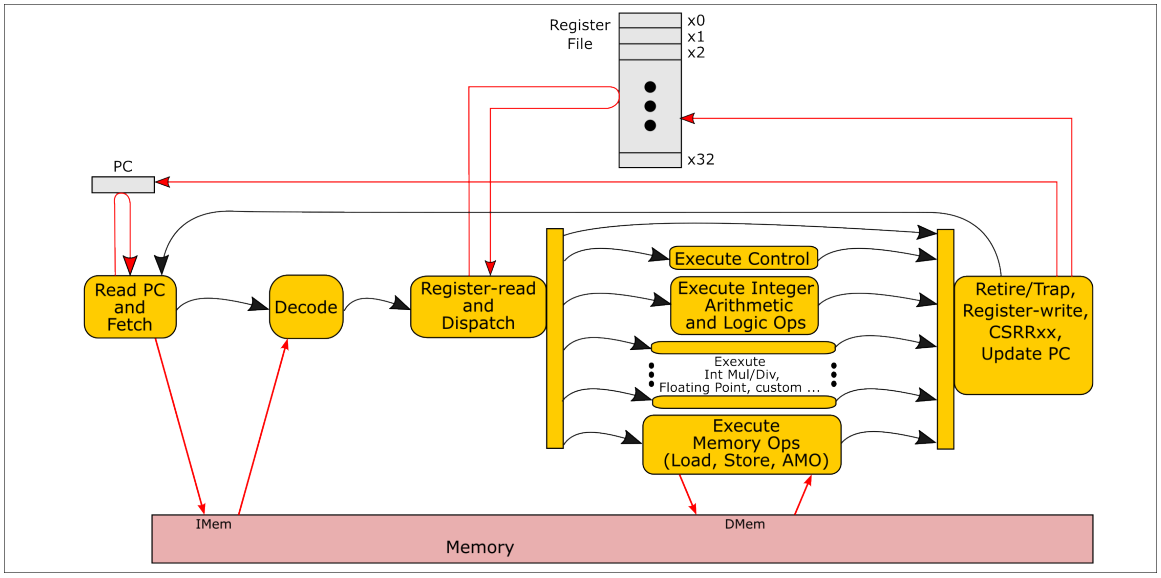

Figure 3.1: Simple interpretation of RISC-V instructions

- 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.

3.3 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).

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, 4.3, 4.4, 4.9, —we will learn the BSV concepts needed to code these ideas.

4.2 BSV: 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)`.

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 BSV: 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).

4.3.1 BSV: 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.

4.3.2 BSV: 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 [14]. 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:

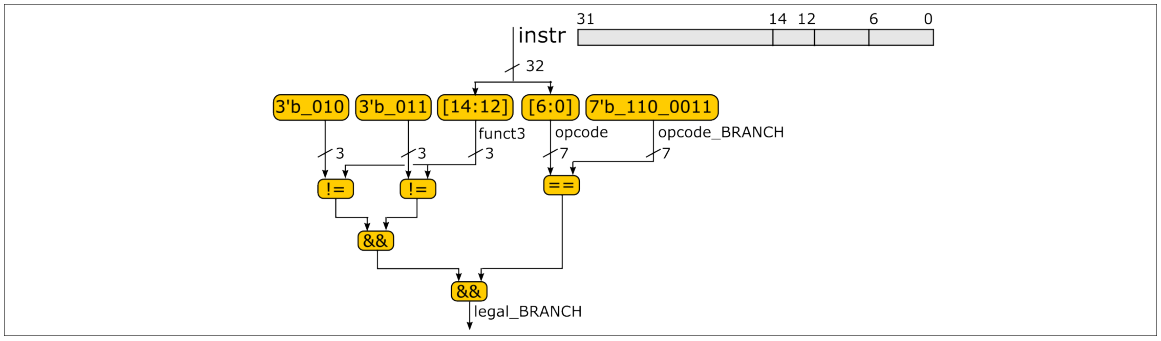

Figure 4.2: Testing for a legal BRANCH instruction

- 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).

4.3.3 BSV: 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 4.9, 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.

BSV: 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 we cannot escape 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.

BSV: Data types identify combinational circuits

In BSV, the output type of any circuit that produces a value will have one of three forms (the latter two will be discussed in detail later):

- *... some value type ...*: `Bit#(n)`, `Bool`, or types discussed later such as structs, enums, tagged unions, vectors, and so on.
- `Action`
- `ActionValue #(` *... some value type ...* `)`

Circuits whose output types are of the first form are *guaranteed* by BSV's type-checking system to be pure combinational circuits—they cannot have any side-effects such as updating a register or memory location, enqueuing or dequeuing from a FIFO, outputting a value to a device or display, *etc.* Circuits with `Action` and `ActionValue#()` types, on the other hand, may have side-effects as part of the process of producing their output value.

BSV is unusual amongst programming languages in precisely tracking "pure" (no side-effects) *vs.* "impure" constructs through type-checking. In this regard it is most similar to Haskell, where potentially impure expressions must have certain monadic types (typically in the "IO monad"). The importance of tracking purity will become clear when we discuss rule and method conditions and scheduling, later.

4.4 BSV: Functions

The fragment of code shown above can be packaged into a BSV function, specifying argument(s) with precise types and a precise result type:

```
1  function Bool is_legal_BRANCH (Bit #(32) instr);
2     Bit #(7) opcode_BRANCH = 7'b_110_0011;
3
4     Bit #(7) opcode = instr [6:0];
5     Bit #(3) funct3 = instr [14:12];
6     Bool legal = ((opcode == opcode_BRANCH)
7                   && (funct3 != 3'b010)
8                   && (funct3 != 3'b011));
9     return legal;
10 endfunction
```

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

```
   Bit #(32) x, y;

   Bool result_x = is_legal_BRANCH (x);
   Bool result_y = is_legal_BRANCH (y);
```

BSV function definition and application syntax is similar to SystemVerilog.

4.5 BSV: A small testbench to test our code

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

```
import StmtFSM :: *;

function Bool is_legal_BRANCH (Bit #(32) instr);
   ... as shown earlier ...
endfunction

(* synthesize *)
module mkTop (Empty);

   mkAutoFSM (
      seq
         action
            Bit #(32) instr_BEQ = {7'h0, 5'h9, 5'h8, 3'b000, 5'h3, 7'b_110_0011};
            $display ("instr_BEQ %08h => %0d", instr_BEQ,
                      is_legal_BRANCH (instr_BEQ));
         endaction

         action
            Bit #(32) instr_BNE = {7'h0, 5'h9, 5'h8, 3'b001, 5'h3, 7'b_110_0011};
            $display ("instr_BNE %08h => ", instr_BNE,
                      fshow (is_legal_BRANCH (instr_BNE)));
         endaction

         action
            Bit #(32) instr_ILL_op = {7'h0, 5'h9, 5'h8, 3'b100, 5'h3, 7'b_110_0000};
            $display ("instr_ILL_op %08h => ", instr_ILL_op,
                      fshow (is_legal_BRANCH (instr_ILL_op)));
         endaction

         action
            Bit #(32) instr_ILL_f3 = {7'h0, 5'h9, 5'h8, 3'b010, 5'h3, 7'b_110_0011};
            $display ("instr_ILL_f3 %08h => %0d", instr_ILL_f3,
                      is_legal_BRANCH (instr_ILL_f3));
         endaction
      endseq);

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 8). 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 [14]. 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_IALU (Bit #(32) instr);
      ... acccepts LUI, AUIPC, ADDI, SLTI, ..., OR, AND

   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.

□

4.6 BSV: `enum` types

In Figure 3.1, in the Register-Read and Dispatch step, we need to know whether the incoming instruction is illegal, 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 illegal, 1 for control, 2 for IALU, 3 for Mem), but it is more readable, and cleaner, to use an "enum" type (similar to enum types in SystemVerilog and C):

```
typedef enum {OPCLASS_ILLEGAL,
              OPCLASS_CONTROL,    // BRANCH, JAL, JALR
              OPCLASS_IALU,
              OPCLASS_MEM }       // LOAD, STORE, AMO
OpClass
deriving (Bits, Eq, FShow);
```

This defines a type `OpClass` containing four constants, `OPCLASS_ILLEGAL`, `OPCLASS_CONTROL`, and so on.

4.6.1 `deriving (Bits)`

Because we said "`deriving(Bits)`", the *bsc* compiler will automatically represent them with the obvious codes 0, 1, 2 and 3 in a minimal bit-width (`Bit#(2)`). For other codings, we would *not* say "`deriving(Bits)`", and we would provide an explicit mapping function into codes (see "typeclass instances", later).

4.6.2 `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. For other definitions of equality/inequality, we would *not* say "`deriving(Eq)`", and we would define equality/inequality as we wish (see "typeclass instances", later).

4.6.3 `deriving (FShow)`

Given a value v of type `OpClass`, if we directly print it (*e.g.,* in a `$display()` statement), it will print its numeric code (0, 1, ...). 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 its symbolic name from the enum declaration instead (*i.e.,* it will print the labels `OPCLASS_ILLEGAL`, `OPCLASS_CONTROL`, and so on).

4.7 BSV: 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 4.6, the identifiers `OPCLASS_ILLEGAL`, `OPCLASS_CONTROL`, `OPCLASS_IALU`, `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.

4.8 BSV: 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.

4.9 BSV: 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 other instructions, returning an `OpClass` (Section 4.6):

```
function OpClass instr_opclass (Bit #(32) instr);
   OpClass result;
   if (is_legal_BRANCH (instr)
      || is_legal_JAL (instr)
      || is_legal_JALR (instr))
      result = OPCLASS_CONTROL;
   else
      result = OPCLASS_ILLEGAL;
   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_ILLEGAL);
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 `instr` argument is fed into the circuits for `is_legal_BRANCH()` (hardware schematic in

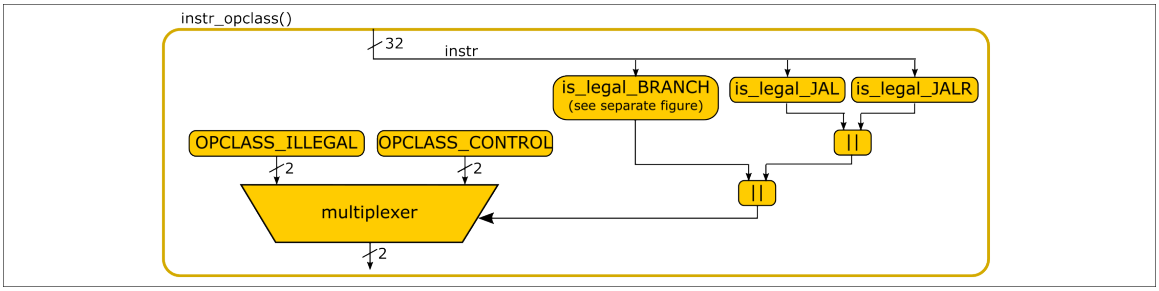

Figure 4.3: If-then-else is a multiplexer

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:

```
1  function Bool instr_opclass (Bit #(32) instr);
2     OpClass result;
3     if (is_legal_BRANCH (instr)
4        || is_legal_JAL (instr)
5        || is_legal_JALR (instr))
6        result = OPCLASS_CONTROL;
7     else if (is_legal_IALU (instr))
8        result = OPCLASS_IALU;
9     else if (is_legal_Mem (instr))
10       result = OPCLASS_MEM;
11    else
12       result = OPCLASS_ILLEGAL;
13    return result;
14 endfunction
```

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

4.9.1 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.

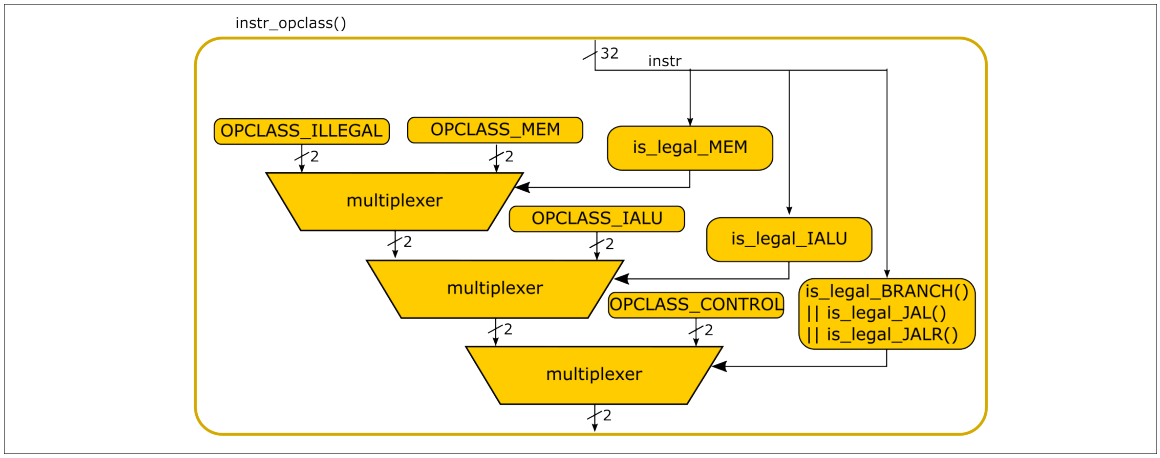

Figure 4.4: Nested if-then-elses become cascaded multiplexers

---

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.

□

4.10 BSV: 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.

4.10.1 BSV: 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]

[2] A limited form of arithmetic is possible on numeric types. Consider a generic function that takes two

4.10.2 BSV: 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.

4.10.3 BSV: 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:

```
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`.

4.10.4 BSV: 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:

---

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.

```
1  `ifdef RV32
2
3     typedef 32 XLEN;
4
5  `elsif RV64
6
7     typedef 64 XLEN;
8
9  `endif
10
11    Integer xlen = valueOf (XLEN);
```

As in Verilog and SystemVerilog, pre-processor directives begin with a ` character (backtick) (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

Struct types (BSV)
Memory requests and responses (RISC-V)

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

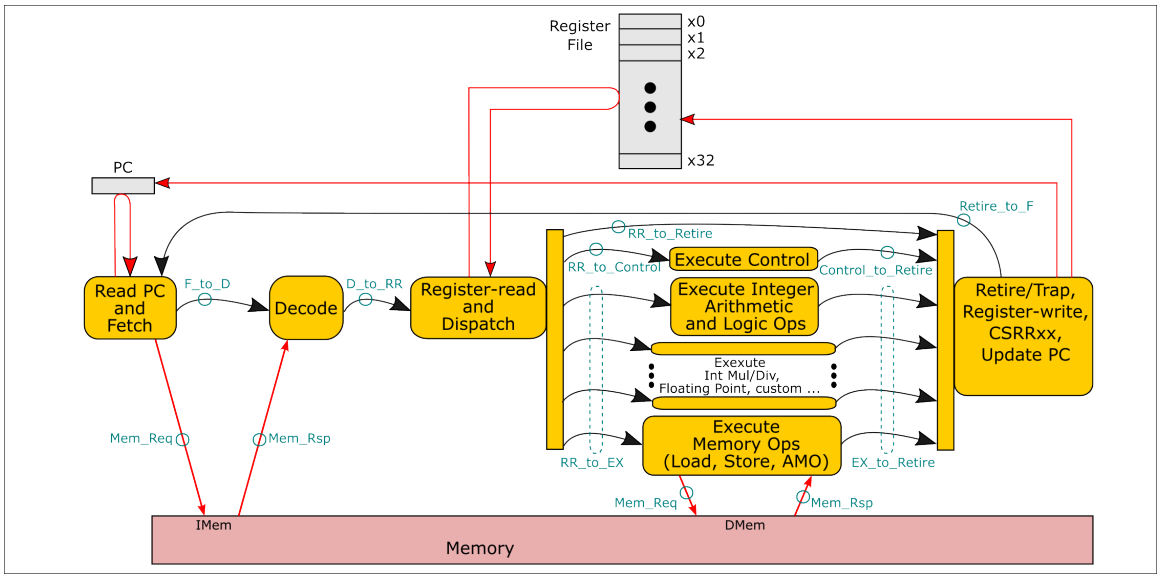

Figure 5.1: Simple interpretation of RISC-V instructions (Fig. 3.1 with arrows annotated with `struct` types)

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 [14], 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:

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

                Bool         exception;
                Bit #(XLEN)  cause;      // Memory exception during Fetch;
                                         // Illegal instr decoded.
                // If not exception
                Bit #(XLEN)  fallthru_pc;
                Bit #(32)    instr;
                OpClass      opclass;
                Bool         has_rs1;
                Bool         has_rs2;
                Bool         has_rd;
                Bit #(32)    imm;          // Canonical (bit-swizzled)
} D_to_RR
deriving (Bits, FShow);
```

Because we said "`deriving(Bits)`", the *bsc* compiler will automatically work out a representation for `D_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 `D_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]

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:

```
D_to_RR {inum=..., pc=..., instr=..., ...  }
```

5.2.1 Creating struct values

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

```
     D_to_RR x = D_to_RR {pc:          ... value of field ... ,
                          exception:   ... value of field ... ,
                          cause:       ... value of field ... ,
                          fallthru_pc: ... value of field ... ,
                          instr:       ... value of field ... ,
                          ...};
```

[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.

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 `D_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 = D_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 = D_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 is either for reading data ("LOAD") or for writing data ("STORE").[3] We can express these request-types using an enum type (similar to enums in C or SystemVerilog):

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

An IMem request is for one 32-bit instruction (four bytes).[4] A DMem request may be for one, two or four bytes.[5] We express these request-size options using an enum type:

```
1 typedef enum {MEM_1B, MEM_2B, MEM_4B} Mem_Req_Size
2 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:

[3] When implementing the "A" RISC-V ISA extension (Atomic Ops), memory requests can also be for LR (Load-Reserved), SC (Store-Conditional), AMOSWAP, AMOADD, AMOXOR, AMOAND, AMOOR, AMOMIN, AMOMAX, AMOMINU, and AMOMAXU.

[4] 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.

[5] In RV64, and with the "D" RISC-V ISA extension for double-precision floating-point even in RV32, memory-requests can also be for eight bytes.

```
typedef struct {Mem_Req_Type  req_type;
                Mem_Req_Size  size;
                Bit #(XLEN)   addr;
                Bit #(XLEN)   data;    // Only for STORE
} Mem_Req
deriving (Eq, FShow, 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,[6] 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 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.

[6] 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.

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:

```
1 typedef enum {MEM_RSP_OK,
2               MEM_RSP_MISALIGNED,
3               MEM_RSP_ERR         } Mem_Rsp_Type
4 deriving (Eq, FShow, Bits);
```

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

```
typedef struct {Mem_Rsp_Type  rsp_type;
                Bit #(XLEN)   data;    // Only for LOAD
} 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

The core functions for RISC-V execution

6.1 Introduction

In this chapter we discuss the core functions of Figure 5.1, which we repeat here for convenience (all the `struct` declarations can be found in the file: `src_Common/Inter_Stage.bsv`).

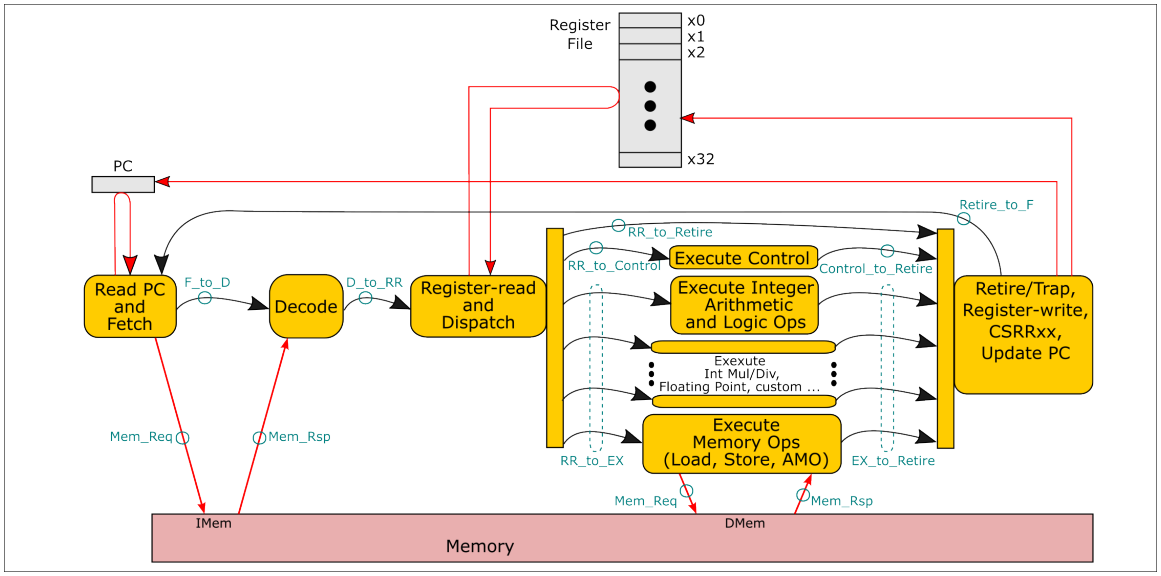

Figure 6.1: Simple interpretation of RISC-V instructions (same as Fig. 3.1, with arrows annotated with `struct` types)

6.2 RISC-V: The Fetch function

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 the previous chapter.
- Some additional information "`F_to_D`" passed on to the Decode step.

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

```
typedef struct {
   Bit #(XLEN) pc;
} F_to_D
deriving (Bits, FShow);
```

NOTE:

BSV: single-field structs
It might seem like overkill to define a struct for just one field like the PC (why not just pass the PC?), but it has the following advantages:

- It becomes easy to add more fields later, should we need to do so. In particular, for Fife we will need to add some branch-prediction information. We may also wish to add temporary fields that aid in debugging.
- Stronger type-checking: each new struct type is distinct from all other types in a BSV program. Thus, the type-checker will catch any error where we may inadvertantly pass some unrelated and irrelevant XLEN-wide value in place of an `F_to_D` struct.
- There is *no* runtime cost: an `F_to_D` value occupies the same XLEN bits as the `pc` field by itself.

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

```
typedef struct {
   F_to_D   to_D;        // info to Decode step
   Mem_Req  mem_req;     // requst info to memory
} Result_F
deriving (Bits, FShow);
```

Finally, as mentioned earlier, the function `fn_F` is almost trivial:

```
function Result_F fn_F (Bit #(XLEN)  pc)
   let y = Result_F {to_D: F_to_D {pc: pc},
                     mem_req: Mem_Req {req_type: MEM_LOAD,
                                       size:     MEM_4B,
                                       addr:     pc,
                                       data :    ?}};
   return y;
endfunction
```

Exercise 6.1:

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

□

6.3 RISC-V: The Decode function

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

```
function D_to_RR
         fn_D (F_to_D  x_F_to_D, Mem_Rsp  rsp_IMem);

   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 = D_to_RR {pc:           x_F_to_D.pc,

                    exception:    False,
                    cause:        ?,

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

   if (rsp_IMem.rsp_type == MEM_RSP_MISALIGNED) begin
      y.exception = True;
      y.cause     = cause_INSTRUCTION_ADDRESS_MISALIGNED;
   end
   else if (rsp_IMem.rsp_type == MEM_RSP_ERR) begin
      y.exception = True;
      y.cause     = cause_INSTRUCTION_ACCESS_FAULT;
   end
   else if (is_legal_LUI (instr) || is_legal_AUIPC (instr)) begin
      y.opclass = OPCLASS_IALU;
      y.has_rd  = non_zero_rd;
      y.imm     = zeroExtend (instr_imm_U (instr));
```

```
37    end
38    else if (is_legal_BRANCH (instr)) begin
39       y.opclass = OPCLASS_CONTROL;
40       y.has_rs1 = True;
41       y.has_rs2 = True;
42       y.imm     = zeroExtend (instr_imm_B (instr));
43    end
44    else if (is_legal_JAL (instr)) begin
45       y.opclass = OPCLASS_CONTROL;
46       y.has_rd  = non_zero_rd;
47       y.imm     = zeroExtend (instr_imm_J (instr));
48    end
49    else if (is_legal_JALR (instr)) begin
50       y.opclass = OPCLASS_CONTROL;
51       y.has_rs1 = True;
52       y.has_rd  = non_zero_rd;
53       y.imm     = zeroExtend (instr_imm_I (instr));
54    end
55    else if (is_legal_LOAD (instr)) begin
56       y.opclass = OPCLASS_MEM;
57       y.has_rs1 = True;
58       y.has_rd  = non_zero_rd;
59       y.imm     = zeroExtend (instr_imm_I (instr));
60    end
61    else if (is_legal_STORE (instr)) begin
62       y.opclass    = OPCLASS_MEM;
63       y.has_rs1    = True;
64       y.has_rs2    = True;
65       y.writes_mem = True;
66       y.imm        = zeroExtend (instr_imm_S (instr));
67    end
68    else if (is_legal_OP_IMM (instr)) begin
69       y.opclass = OPCLASS_IALU;
70       y.has_rs1 = True;
71       y.has_rd  = non_zero_rd;
72       y.imm     = zeroExtend (instr_imm_I (instr));
73    end
74    else if (is_legal_OP (instr)) begin
75       y.opclass = OPCLASS_IALU;
76       y.has_rs1 = True;
77       y.has_rs2 = True;
78       y.has_rd  = non_zero_rd;
79    end
80    else if (is_legal_ECALL (instr) || is_legal_EBREAK (instr)) begin
81       y.opclass = OPCLASS_SYSTEM;
82    end
83    else if (is_legal_FENCE (instr) || is_legal_FENCE_I (instr)) begin
84       y.opclass = OPCLASS_FENCE;
85       y.has_rs1 = True;
86       y.has_rd  = non_zero_rd;
87    end
88    else begin
89       y.exception = True;
90       y.cause     = cause_ILLEGAL_INSTRUCTION;
```

```
91      end
92
93      return y;
94  endfunction
```

In line 4, we extract the instruction from the `Mem_Rsp` memory response from the Fetch operation, and in line 5 we extract the `rd` ("destination register") field from the instruction.

In line 4, the `truncate` operation is used to shrink the bit-vector width of `rsp_IMem.data` to the bit-vector width of `instr` (32-bits). But why is this shrinkage needed? After all, in the declaration of the `Mem_Rsp` type, the `data` field is declared to have type `Bit#(XLEN)` which is synonymous with `Bit#(32)`. The reason is that we've written the code to be ready for re-use in the future with RV64, when `XLEN` will be defined as 64. The `truncate` operation is polymorphic, accepting arguments of any bitwidth wider than the required output. Here, when the argument is 32-bits wide, it does nothing (is a no-op); but for RV64, when the argument is 64-bits wide, it performs the shrinkage to 32 bits. Note, `truncate` keeps least-significant bits and drops most-significant bits.

In line 7 we compute the fall-through PC, PC+4 (with the caveat that if we want 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 10-23 we create a baseline `D_to_RR` value which we will selectively modify in the if-then-elses below.

In lines 25-32 we first handle the sitation where the Fetch operation to memory itself returned an error. We mark the `exception` field True and fill in the appropriate `cause`.

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 [14].

Each if-then-else clause also fills in the `has_rs1`, `has_rs2` and `has_rd` fields, as appropriate, for each class of instruction. We also decode each kind of "immediate" field and fill in the `imm` field.

The final "else" clause (lines 88-91) is evaluated if the instruction does not match one of the forty RV32I instructions. In this case we set the `exception` and `cause` fields to indicate an illegal instruction.

Note the use of functions `instr_imm_I()`, `instr_imm_S()`, `instr_imm_B()`, `instr_imm_U()` and `instr_imm_J()`, which extract the "immediate" field from a 32-bit instruction. For the different classes of instruction (I, S, B, U, J), the immediates are coded differently, and have different widths (see the schemata at the top of the RV32I page in "Table 24.2: Instruction listing for RISC-V" of the Unprivileged Spec [14]). These functions decode the bits and zero-extend them to XLEN bits (the width of `d_to_rr.imm`). Later, in the "execute" code for each instruction, we will pick out the relevant bits out of the XLEN bits, and zero-extend or sign-extend them as needed for the particular instruction.

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

Observe that the entire `fn_D()` 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_D()` function can be visualized as a box with incoming wires corresponding to `F_to_D` and `Mem_Rsp`, outgoing wires corresponding to `D_to_RR`, and filled with logic gates that compute each output wire as a function of the input wires.

Exercise 6.2:

Write the functions `instr_imm_I()`, `instr_imm_S()`, `instr_imm_B()`, `instr_imm_U()` and `instr_imm_J()`.

Exercise 6.3:

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

□

6.4 RISC-V: Register Read and the Dispatch function ("RRD")

In the Register-Read and Dispatch step, we read the `rs1` and `rs2` values from the Register File, and then use a Dispatch function to determine what needs to be passed to the the alternative following steps.

We will cover register files in Section 7.4 register-reads in Chapter 8.

The core function for Dispatch step is called `fn_Dispatch`, and is rather simple. Its

The argument for `fn_RR` is a `D_to_RR` struct from the Decode step, which was described in Section 5.2. Before we look at its result type let us look at the "flows" through the subsequent steps.

There are four possible "flows" after RR:

- Some information is sent directly from RR to Retire for *every* instruction. This includes any error indication (memory error during Fetch, or decode illegal instruction).

  Equally important, RR sends a *tag* to Retire indicating which, if any, of the next three bits of information have also been produced.

- If there is no error indication, then we also perform one of the following three flows:
  - If the instruction is a BRANCH, JAL or JALR, we produce information to Execute Control.
  - If the instruction is a LUI, AUIPC or integer arithmetic or logic (IALU) operation, we produce information to Execute Integer Arithmetic and Logic Ops.
  - If the instruction is a LOAD or STORE, we produce information to Execute Memory Ops.

The following type, `Exec_Tag`, identifies which of the four flows is being performed for this instruction:

```
typedef enum {EXEC_TAG_RETIRE,     // to Retire only
              EXEC_TAG_CONTROL,    // to Retire and Control
              EXEC_TAG_IALU,       // to Retire and IALU
              EXEC_TAG_DMEM        // to Retire and DMem
} Exec_Tag
deriving (Bits, Eq, FShow);
```

The information sent to Retire directly is defined in the type `RR_to_Retire`:

```
typedef struct {
                // Informs Retire about flow for this instruction
                Exec_Tag     exec_tag;

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

                // If not exception
                Bool         has_rd;
                Bit #(5)     rd;
                Bit #(XLEN)  fallthru_pc;

} RR_to_Retire
deriving (Bits, FShow);
```

The `exec_tag` informs Retire about the flow for this instruction. The `exception` and `cause` fields are carried through since it is the Retire step that handles traps. Note, in addition to passing exceptions from RR to Retire, the Control or Execute steps could also raise 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` and `rd` fields are 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).

It's result type, `Result_RR`, is just a nested struct that contains the three different struct types shown in Figure 8.1:

The `RR_to_Control` and `RR_to_EX` types were described in Section 6.5 and Section 6.6, respectively.

We can now describe the result of the `fn_RR` function, which is simply a nested struct containing the previously described structs:

```
typedef struct {
   RR_to_Retire   to_Retire;    // to Retire only
   RR_to_Control  to_Control;   // to Retire and to Control
   RR_to_EX       to_EX;        // to Retire and one of the execute pipes
} Result_Dispatch
deriving (FShow);
```

The first component goes directly to the Retire step. The second component goes to the Control step. The third component is used for the Execute Integer Arithmetic Ops step and for the Execute DMem Ops steps.

The code for `fn_RR` is shown below. It's arguments are the interface to the register file and the information from the Decode step:

```
function Result_Dispatch fn_Dispatch (D_to_RR             x,
                                      Bit #(XLEN)         rs1_val,
                                      Bit #(XLEN)         rs2_val);
   // Compute tag to control merging at Retire
   Exec_Tag exec_tag = EXEC_TAG_RETIRE;    // exceptions and OPCLASS_SYSTEM
   if (! x.exception) begin
      if (is_BRANCH (x.instr)
         || is_JAL (x.instr)
         || is_JALR (x.instr))             exec_tag = EXEC_TAG_CONTROL;
      else if (x.opclass == OPCLASS_IALU)  exec_tag = EXEC_TAG_IALU;
      else if (x.opclass == OPCLASS_MEM)   exec_tag = EXEC_TAG_DMEM;
   end

   // ----------------
   // Info direct for Retire
   let to_Retire = RR_to_Retire {exec_tag:     exec_tag,

                                 exception:    x.exception,
                                 cause:        x.cause,
                                 pc:           x.pc,

                                 has_rd:       x.has_rd,
                                 rd:           x.rd
                                 fallthru_pc:  x.fallthru_pc};

   // ----------------
   // Info for Control
   let to_Control = RR_to_Control {pc:           x.pc,
                                   fallthru_pc:  x.fallthru_pc,
                                   instr:        x.instr,
                                   rs1_val:      rs1_val,
                                   rs2_val:      rs2_val,
                                   imm:          x.imm};

   // ----------------
   // Info for Execute pipes
   let to_EX  = RR_to_EX {instr:   x.instr,
                          rs1_val: rs1_val,
                          rs2_val: rs2_val,
                          imm:     x.imm};

   // ----------------
   let result = Result_Dispatch {to_Retire:  to_Retire,
                                 to_Control: to_Control,
                                 to_EX:      to_EX};
   return result;
```

```
47 endfunction
```

Lines 2-9 read the `rs1` and `rs2` values from the register file. Note that the `rs1` and `rs2` fields are not meaningful for all instructions; in those cases, these reads may be "wild" reads from random registers. This does not matter, because the values will only be used if they are needed.

Lines 13-20 compute the `exec_tag` based on the kind of insrruction.

Lines 24-32, 36-41 and 45-48 construct the information for the Retire, Control, and IALU/DMem steps, respectively. Like the register-read discussion above described above, the latter two are meaningful only for certain instructions, but we can construct them anyway; they will only be used if meaningful.

Lines 51-54 construct the final result and return it.

6.5 RISC-V: The Execute Control function

We will next discuss the "execute" functions (Control, IALU and DMem), which can also be characterized as pure value-to-value functions.

The input and output of `fn_Control`, the Control function in Figure 5.1, are values of the following types, respectively. Each of the input fields is needed to compute one or more of the output fields.

```
1  typedef struct {Bit #(XLEN)  pc;
2                  Bit #(XLEN)  fallthru_pc;
3                  Bit #(32)    instr;
4                  Bit #(XLEN)  rs1_val;
5                  Bit #(XLEN)  rs2_val;
6                  Bit #(32)    imm;
7  } RR_to_Control
8  deriving (Bits, Eq, FShow);
9
10 typedef struct {Bool         exception;
11                 Bit #(XLEN)  cause;        // Misaligned BRANCH/JAL/JALR target
12
13                 Bit #(XLEN)  next_pc;
14                 Bool         data_valid;    // True for JAL/JALR; False for BRANCH
15                 Bit #(XLEN)  data;          // Return-PC for JAL/JALR
16 } Control_to_Retire
17 deriving (Bits, FShow);
```

Here is the Control function `fn_Control`:

```
1  function Control_to_Retire fn_Control (RR_to_Control  x);
2     let instr   = x.instr;
3     let rs1_val = x.rs1_val;
4     let rs2_val = x.rs2_val;
5
```

```
6     Bit #(XLEN)  next_pc   = ?;
7     Bool         exception = False;    // Misaligned target_pc
8
9     if (is_BRANCH (instr)) begin
10       Bool branch_taken = case (instr_funct3 (instr))
11                              funct3_BEQ:  (rs1_val == rs2_val);
12                              funct3_BNE:  (rs1_val != rs2_val);
13                              funct3_BLT:  signedLT (rs1_val, rs2_val);
14                              funct3_BGE:  signedGE (rs1_val, rs2_val);
15                              funct3_BLTU: (rs1_val < rs2_val);
16                              funct3_BGEU: (rs1_val >= rs2_val);
17                            endcase;
18       Bit #(13) imm13 = x.imm [12:0];
19       let target_pc = x.pc + signExtend (imm13);
20       next_pc = (branch_taken ? target_pc : x.fallthru_pc);
21       exception = (branch_taken && (target_pc [1:0] != 0));
22    end
23    else if (is_JAL (instr)) begin
24       Bit #(21) imm21 = x.imm [20:0];
25       next_pc = x.pc + signExtend (imm21);
26       exception = (next_pc [1:0] != 0);
27    end
28    else if (is_JALR (instr)) begin
29       Bit #(12) imm12 = x.imm [11:0];
30       // zero out LSB in target PC
31       next_pc = ((rs1_val + signExtend (imm12)) & ~1);
32       exception = (next_pc [1:0] != 0);
33    end
34
35    Bool data_valid = ((instr_rd  (instr) != 0)
36                       && (is_JAL (instr) || is_JALR (instr)));
37    let y = Control_to_Retire {inum:       x.inum,
38                               pc:         x.pc,
39                               instr:      x.instr,
40                               exception:  exception,
41                               cause:      cause_INSTRUCTION_ADDRESS_MISALIGNED,
42                               next_pc:    next_pc,
43                               data_valid: data_valid,
44                               data:       x.fallthru_pc};
45    return y;
46 endfunction
```

Lines 9-22 handle BRANCH (conditional branch) instructions. First the `case` expression computes the boolean value `branch_taken`, the decision whether to take the branch or not. This is based on the 3-bit `funct3` field of the instruction that identifies the specific condition to be tested. Note that for BLT and BGE, we use the `signedLT` and `signedGE` functions that interpret `rs1_val` and `rs2_val` as signed integers.

Line 19 computes the target PC should the the branch be taken. Line 20 computes the next PC, which is either the target PC or the fall-through PC depending on whether the branch is taken or not. Finally, Line 21 checkes that if the branch is taken, that the target PC is a suitably aligned address.

Lines 23-27 handle JAL (Jump and Link) instructions, and lines 28-33 handle 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.

Notice that the nested if-then-else has no final "else" clause. This is safe because of the checks already done in the Decode function `Fn_D`, which guarantee that this function will only be invoked with inputs that are handled by one of the three clauses.

Finally, lines 35 through 45 construct the final result and return it.

NOTE:

RISC-V: misaligned branch/jump targets

In `fn_Control`, the BRANCH, JAL and JALR clauses set the `exception` field to true if the next PC is not aligned. This is required by the RISC-V Spec (see "Section 2.5 Control Transfer Instructions" in [14]). If there is a misalignment error, we encounter it here on the control-transfer instruction.

If we do not check alignment here, we will encounter a misalignment error on the next Fetch, at the next-PC address. The choice of catching this earlier, at the control-transfer instruction itself, is a design choice by the RISC-V ISA architects.

Exercise 6.4:

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

Exercise 6.5:

In `Fn_Control`, can we change the final "if" condition line:

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

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

```
else begin
```

What might be the hardware implication of such a change?

Exercise 6.6:

In line 35 in `fn_Control`, we are testing `(instr_rd (instr) != 0)`. But we tested this in computing the boolean `has_rd` in `fn_D`, the Decode function. Discuss the hardware tradeoffs in just passing that boolean value along in the structs to this point, instead of recomputing it here.

□

6.6 RISC-V: The Execute Integer Ops function

The input and output of `fn_EX_IALU`, the "Execute Integer Ops" function in Figure 5.1, are values of the following types, respectively. Each of the input fields is needed to compute one or more of the output fields.

```
1  typedef struct {Bit #(XLEN)  pc;
2                  Bit #(32)    instr;
3                  Bit #(XLEN)  rs1_val;
4                  Bit #(XLEN)  rs2_val;
5                  Bit #(32)    imm;
6  } RR_to_EX
7  deriving (Bits, FShow);
8
9  typedef struct {Bool         exception;
10                 Bit #(XLEN)  cause;
11
12                 Bool         data_valid;
13                 Bit #(XLEN)  data;
14 } EX_to_Retire
15 deriving (Bits, FShow);
```

Here is the Execute Integer Ops function `fn_EX_IALU`:

```
1  function EX_to_Retire fn_EX_IALU (RR_to_EX x);
2     let instr = x.instr;
3
4     let y = EX_to_Retire {inum:       x.inum,
5                           pc:         x.pc,
6                           instr:      instr,
7                           exception:  False,
8                           cause:      ?,
9                           data_valid: True,
10                          data:       ?};
11
12    if (is_LUI (instr))
13       y.data = (zeroExtend (instr_imm_U (instr)) << 12);
14    else if (is_AUIPC (instr)) begin
15       Bit #(XLEN) offset = signExtend ({ instr_imm_U (instr), 12'b0 });
16       y.data = x.pc + offset;
17    end
18    else begin
19       let result <- fn_IALU (logf, instr, x.rs1_val, x.rs2_val, x.imm);
20       y.data = result;
21    end
22    return y;
23 endfunction
```

Lines 12-13 handle LUI instructions (Load Upper Immediate). Lines 14-17 handle AUIPC instructions (Add Upper Immediate to PC). Lines 18-21 invoke `fn_IALU` to perform all the remaining Integer ops, and this is shown in the code below.

```
1  function Bit #(XLEN) fn_IALU (Bit #(32)    instr,
2                                Bit #(XLEN)  v1,
3                                Bit #(XLEN)  v2,
4                                Bit #(32)    imm);
5     Bit #(7)    opcode = instr_opcode (instr);
6     Bit #(3)    funct3 = instr_funct3 (instr);
7     Int #(XLEN) iv1    = unpack (v1);
8     Int #(XLEN) iv2    = unpack (v2);
9     Int #(XLEN) i_imm  = unpack (signExtend (instr_imm_I (instr)));
10
11    Bit #(XLEN) y_OP     = 0;
12    if (opcode == opcode_OP) begin
13       Bit #(5) shamt = v2 [4:0];
14       case (funct3)
15          funct3_ADD:  y_OP = pack ((instr [30] == 1'b0)
16                                    ? (iv1 + iv2)
17                                    : (iv1 - iv2));
18          funct3_SLL:  y_OP = v1 << shamt;
19          funct3_SLT:  y_OP = ((iv1 < iv2) ? 1 : 0);
20          funct3_SLTU: y_OP = ((v1  < v2)  ? 1 : 0);
21          funct3_XOR:  y_OP = v1 ^ v2;
22          funct3_SRL:  y_OP = v1 >> shamt;
23          funct3_SRA:  y_OP = pack (iv1 >> shamt);
24          funct3_OR:   y_OP = v1 | v2;
25          funct3_AND:  y_OP = v1 & v2;
26       endcase
27    end
28
29    Bit #(XLEN) y_OP_IMM = 0;
30    if (opcode == opcode_OP_IMM) begin
31       Bit #(5) shamt = imm [4:0];
32       case (funct3)
33          funct3_ADDI:  y_OP_IMM = pack (iv1 + i_imm);
34          funct3_SLTI:  y_OP_IMM = ((iv1 < i_imm) ? 1 : 0);
35          funct3_SLTIU: y_OP_IMM = ((v1  < imm)   ? 1 : 0);
36          funct3_XORI:  y_OP_IMM = v1 ^ imm;
37          funct3_ORI:   y_OP_IMM = v1 | imm;
38          funct3_ANDI:  y_OP_IMM = v1 & imm;
39          funct3_SLLI:  y_OP_IMM = v1 << shamt;
40          funct3_SRLI:  y_OP_IMM = v1 >> shamt;
41          funct3_SRAI:  y_OP_IMM = pack (iv1 >> shamt);
42       endcase
43    end
44
45    return (y_OP | y_OP_IMM);
46 endfunction
```

In lines 7-8, we define signed-integer versions `iv1`, `iv2` of the unsigned integer values `v1` and `v2`, respectively. There is no hardware cost to this definition, it's simply a declaration 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 19-20 compute the SLT (Set Less Than (signed)) and SLTU (Set Less Than Unsigned) operations. The

SLT op 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 13 extracts a 5-bit "shift amount" from the `rs2` value for the shift operators SLL, SRL and SRA. Line 31 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 in at the most-significant bit side are zero (SRL) or replicate the most-significant bit (SRA). SRLI and SRAI exhibit a similar difference.

In lines 23 (SRA) and 41 (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 11-27 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`. It defaults to 0 when it is not an `op_OP`.

Lines 29-43 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. It 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 6.7:

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 4.9.1 on parallel and sequential multiplexers (mux). Note: in our code, we have explicitly coded a parallel mux.

Exercise 6.8:

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

Exercise 6.9:

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

Exercise 6.10:

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 RISC-V: The Execute DMem function

The types of the input and output of the "Execute Memory Ops" function in Figure 5.1 are the same as for "Execute Integer Ops", *i.e.,* `RR_to_EX` and `EX_to_Retire`, which were described in Section 6.6.

Figure 6.2 shows that the "Execute Memory Ops" function can be split into two phases, just like we did for Fetch. The first phase generates a memory request, and the second

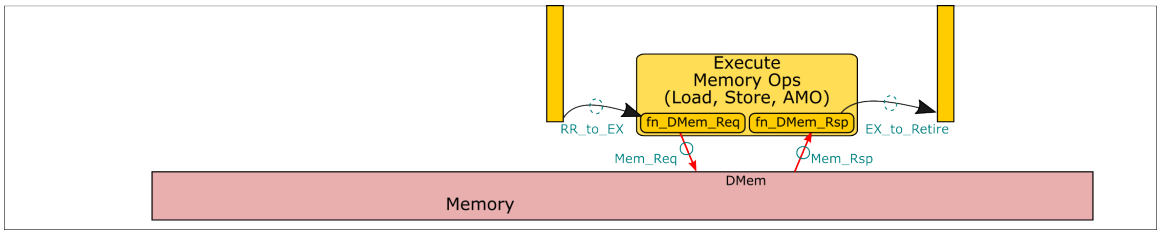

Figure 6.2: The Execute Memory Ops function is split into two phases.

phase processes the memory response. As with Fetch, there is no guarantee how "soon" the response will come.

Here is the function for the first phase, computing a memory request:

```
1  function Mem_Req fn_DMem_Req (RR_to_EX  x);
2     Bool is_LD = is_LOAD  (x.instr);
3     Bool is_ST = is_STORE (x.instr);
4
5     // Mem effective-address calculation
6     Int #(XLEN) ibase   = unpack (x.rs1_val);                  // Signed
7     Int #(XLEN) ioffset = unpack (signExtend (x.imm [11:0]));  // Signed
8     Bit #(XLEN) eaddr   = pack (ibase + ioffset);
9
10    Mem_Req_Size mrq_size = unpack (x.instr [13:12]);  // B, H, W or D
11    Mem_Req_Type mrq_type = (is_LD ? funct5_LOAD : funct5_STORE);
12
```

```
13    let y = Mem_Req {inum:     x.inum,
14                     pc:       x.pc,
15                     instr:    x.instr,
16                     req_type: mrq_type,
17                     size:     mrq_size,
18                     addr:     zeroExtend (eaddr),
19                     data :    zeroExtend (x.rs2_val)};
20    return y;
21 endfunction
```

Lines 6-8 compute the so-called "Effective Address" of a LOAD/STORE, by performing a *signed* addition of the `rs1` and immediate values.

Line 10 defines the memory-request size (1, 2, 4 or 8 bytes), which in RISC-V terminology are referred to as Bytes (B), Halfwords (H), Words (W) and Doublewords (D), respectively). The `Mem_Req_Size` type is defined in the file `Mem_Req_Rsp.bsv` as follows:

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

Line 10 relies on the fact that these will be coded as 2'b00, 2'b01 and 2'b10, which is exactly the coding found in `instr[13:12]`, so we can simply "`unpack`" the bits to the type `Mem_Req_Size` without any further manipulation.

Exercise 6.11:

In Line 10, what if `instr[13:12]` had the value 2'b11? This is a legal instruction in the RV64I ISA, for LOADs and STOREs of 8-byte values, but illegal in RV32I. Should we check that here?

Hint: study the `is_legal_LOAD()` `is_legal_STORE()` functions used in the Decode function `fn_D()` to see if we will ever encounter an illegal value here.

Exercise 6.12:

Write a different version of Line 10 that does not rely on the one-to-one coding equivalence of `instr[13:12]` and the bit coding of `Mem_Req_Size`.

Hint: The solution will be a nested if-then-else, or a `case` expression.

□

Here is the function for the second phase, accepting a memory response as argument and computing the struct to be sent to the Retire step.

```
1  function EX_to_Retire fn_DMem_Rsp (Mem_Rsp x);
2     Bool exception = ((x.rsp_type == MEM_RSP_ERR)
3                       || (x.rsp_type == MEM_RSP_MISALIGNED));
4     Bit #(XLEN)  cause = ((x.rsp_type == MEM_RSP_MISALIGNED)
5                                          ? (is_LOAD (x.instr)
6                               ? cause_LOAD_ADDRESS_MISALIGNED
```

```
7                                : cause_STORE_AMO_ADDRESS_MISALIGNED)
8                              : (is_LOAD (x.instr)
9                                 ? cause_LOAD_ACCESS_FAULT
10                                : cause_STORE_AMO_ACCESS_FAULT));
11
12      let y = EX_to_Retire {exception:  exception,
13                            cause:      cause,
14
15                            data_valid: (! is_STORE (x.instr)),
16                            data:       truncate (x.data)};
17      return y;
18   endfunction
```

Lines 2-10 convert the memory-systems "exception" codes (`MEM_RSP_ERR` and `MEM_RSP_MISALIGNED`) into RISC-V `cause` codes (`cause_...`).)

Lines 12-16 constructs the required `EX_to_Retire` result and line 17 returns it.

Line 15 uses `(! is_STORE())` to indicate whether the `data` field is valid or not, *i.e.,* if it is not a STORE, it must be a LOAD, returning data. Note that the code may set `data_valid` to true when there is an exception in a LOAD, but in that case the `data_valid` value does not matter.

Exercise 6.13:

When implementing the "A" RISC-V ISA Extension (Atomic Memory Ops), the repertoire of memory operations widens from LOAD and STORE to include LR (Load-Reserved), SC (Store-Conditional) and a variety of AMOxxx ops such as AMOADD, AMOSWAP, and so on. Will line 15's `(! is_STORE())` remain adequate, in that case?

□

6.8 RISC-V: The Retire function

As seen in Figure 6.1 the Retire function takes an input of type `RR_to_Retire` from Register-Read-and-Dispatch (RRD). This input exists for every instruction. Depending on the kind of instruction, it may also receive an input:

- from Control (of type `Control_to_Retire`), discussed in Section 6.5,
- from Execute Integer Arithmetic and Logic Ops (EXI, of type `EX_to_Retire`), discussed in Section 6.6, or
- from Execute Memory Ops (DMem, also of type `EX_to_Retire`).

One of the outputs of the Retire function, also called the *redirection* output, is an indication of the next PC from which Fetch should resume. This could be:

- the fall-through PC (the most common case); or
- the target PC of a taken BRANCH, or of JAL/JALR; or
- the PC of the trap-vector, in case the current instruction had an exception; or

- the PC of the interrupt handler, in case there is an interrupt pending and we choose to handle it now.

This redirection information from the Retire function is carried in this struct:

```
typedef struct {
    Bit #(XLEN) next_pc;
} F_from_Retire
deriving (Bits, FShow);
```

Note, in Fife, the Fetch unit would have predicted the next PC and already fetched an instruction from that predicted address. If it predicted correctly, the Fetch unit does not need to be redirected from the Retire unit, and we can just discard this information. We'll discuss this in more detail when we discuss Fife in later chapters.

A second output from Retire is an indication of a value to be written back to the GPRs, for those instructions that have an `rd` destination field and which have completed successfully (without an exception).

```
typedef struct {Bool        commit;    // True: write rd
                Bit #(5)    rd;
                Bit #(XLEN) data;
} RW_from_Retire
deriving (Bits, FShow);
```

The `commit` field tells us whether to write a register or not. If true, the other two fields specify the register and the value to be written.

Now we can describe the overall output of Retire, `Result_Retire`:

```
typedef struct {
   Bool         exception;
   Bit #(XLEN)  cause;
   Bit #(XLEN)  exception_pc;

   F_from_Retire   to_F;
   RW_from_Retire  to_RW;
} Result_Retire
deriving (Bits, FShow);
```

If there was an exception in this instruction, `exception` is true, `cause` specifies the kind of exception, and `exception_pc` is the PC of this instruction, to be saved as information for the exception handler. If `exception` is false, then `to_F` specifies the redirection, and `to_RW` specifies the optional register-write.

We can now see the Retire function, `fn_Retire`:

```
function Result_Retire
         fn_Retire (RR_to_Retire         x_RR,
                    Control_to_Retire    x_Control,
```

```
4                       EX_to_Retire         x_EX);
5
6     Bool         exception    = False;
7     Bit #(XLEN)  cause        = ?;
8     Bit #(XLEN)  exception_pc = x_RR.pc;
9     let          y_to_F       = F_from_Retire {next_pc: ?};
10    let          y_to_RW      = RW_from_Retire {commit: False, rd: ?, data: ?};
11
12    // Fill in fields according to the various incoming flows
13    if (x_RR.exec_tag == EXEC_TAG_RETIRE) begin
14       if (x_RR.exception) begin
15          exception = True;
16          cause     = x_RR.cause;
17       end
18       else
19          y_to_F.next_pc = x_RR.fallthru_pc;
20    end
21    else if (x_RR.exec_tag == EXEC_TAG_CONTROL) begin
22       if (x_Control.exception) begin
23          exception = True;
24          cause     = x_Control.cause;
25       end
26       else begin
27          y_to_F.next_pc = x_Control.next_pc;
28          y_to_RW.commit = x_Control.data_valid;
29          y_to_RW.rd     = instr_rd (x_RR.instr);
30          y_to_RW.data   = x_Control.data;
31       end
32    end
33    else begin // exec_tag == EXEC_TAG_IALU/EXEC_TAG_DMEM
34       if (x_EX.exception) begin
35          exception = True;
36          cause     = x_EX.cause;
37       end
38       else begin
39          y_to_F.next_pc = x_RR.fallthru_pc;
40          y_to_RW.commit = x_EX.data_valid;
41          y_to_RW.rd     = instr_rd (x_RR.instr);
42          y_to_RW.data   = x_EX.data;
43       end
44    end
45
46    // Construct and return final result
47    let y = Result_Retire {exception:    exception,
48                           cause:        cause,
49                           exception_pc: exception_pc,
50                           to_F:         y_to_F,
51                           to_RW:        y_to_RW};
52    return y;
53 endfunction
```

The meat of the code is lines 13-44, which is a big if-then-else that examines the incoming `x_RR.exec_tag` and the various exception fields `x_RR.exception`, `x_Control.exception`

and `x_EX.exception` to decide how to fill in the output struct fields.

Exercise 6.14:

In line 33 (the final `else` clause) we have a comment asserting that `exec_tag` must be `EXEC_TAG_IALU` or `EXEC_TAG_DMEM`. Prove this assertion by reasoning forward from the Register-Read-and-Dispatch function `fn_RR`.

□

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 BSV: 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 BSV: 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 can be considered to be the independent, concurrent, internal behavior of a module.

7.2.2 BSV: 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 BSV: 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 BSV: 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 BSV lib: `Reg#(t)`, the register interface

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 BSV: 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 BSV lib: `mkReg(`v`)`, a register module (consructor)

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 BSV: 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
```

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.

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

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 BSV lib: The register file interface: `RegFile#(index_t,data_t)`

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 BSV lib: `mkRegFileFull`, a register file module (constructor)

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.

7.4.3 RISC-V: A register file for RISC-V, with special 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, as follows:

```
1  interface RISCV_GPRs_IFC #(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
6
7  module mkRISCV_GPRs (RISCV_GPRs_IFC #(xlen));
8     RegFile #(Bit #(5), Bit #(xlen)) rf <- mkRegFileFull;
9
10    method Bit #(xlen) read_rs1 (Bit #(5) rs1);
11       return ((rs1 == 0) ? 0 : rf.sub (rs1));
12    endmethod
13
14    method Bit #(xlen) read_rs2 (Bit #(5) rs2);
15       return ((rs2 == 0) ? 0 : rf.sub (rs2));
16    endmethod
17
18    method Action write_rd (Bit #(5) rd, Bit #(xlen) rd_val);
19       rf.upd (rd, rd_val);
20    endmethod
21 endmodule
```

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).

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 7.1: 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 7.2: In `mkRISCV_GPRs` suppose we use `mkRegFile(1,31)` instead of `mkRegFileFull`. What needs to change to accommodate this?

□

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 BSV lib:`FIFOF#(t)`, the FIFO interface

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 BSV lib: `mkFIFOF`, a FIFO module (constructor)

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 ≤ 1 (zero or one element in the queue).

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 BSV: 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 7.3:

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

□

7.5.6 BSV: 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).

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.

NOTE: 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".

7.6 BSV: 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 7.4:

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 BSV: 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 7.4.3. We can directly instantiate this module in a CPU module:

```
1 module mkCPU;
2    ...
3    RISCV_GPRs_IFC #(XLEN) gprs <- mkRISCV_GPRs;
4    ...
5 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 7.5:

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

Exercise 7.6:

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

BSV: FSMs;
RISC-V: the Drum unpipelined CPU

8.1 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 8.1 represents a *processs*, a behavior that evolves over time. For example the Drum CPU

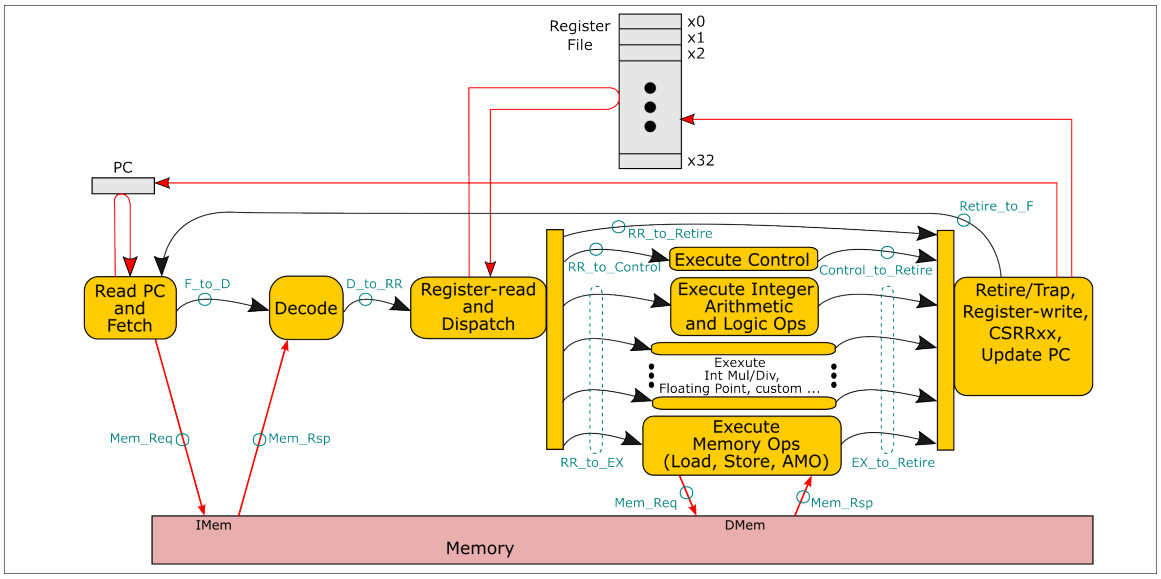

Figure 8.1: Simple interpretation of RISC-V instructions (same as Fig. 6.1)

executes one full instruction after another, and the black arrows in the diagram represent an infinite loop. 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 is the FSM (Finite State Machine). Figure 8.1 can be interpreted as an FSM: each yellow rectangle is a state, and the process transitions from state to state, thereby executing RISC-V instructions. This is exactly what the Drum CPU does. In this chapter, after first discussing FSMs in BSV, we discuss the Drum CPU and its FSM implementation. By the end of this chapter, we will have a complete Drum CPU that is capabable of executing RV32I RISC-V programs.

8.2 BSV: Finite State Machines (FSMs)

Finite state machines (FSMs) are a classical concept in digital hardware design, representing a hardware *process*. An FSM is an artefact that can, at any time, be in one of a number of possible *states*. One state is often distinguised as the *start* state, the initial state of the FSM. From each state, an FSM can *transition* to another state; the choice of destination state may depend on predicates on the current state and on external inputs to the FSM.

One classical notation for FSMs is the "bubble-and-arrow" diagram: a bubble represent a state, and arrows between bubbles represent transitions between states. Thus, an FSM is a specification of a process that, over time, moves from state to to state. The process can loop infinitely, with transitions back to earlier states.

In classical notation, arrows may be annotated with *c/o* labels (conditions and outputs). The condition on an arrow indicates under what conditions this transition is taken, usually a (pure) boolean predicate on the current state and external inputs. A state may have arrows to multiple next-states, with mutually exclusive conditions, thus expressing an if-then-else situation.[1] The outputs on an arrow represent external outputs driven by the FSM starting with that transition and until the next transition. An arrow without a condition is an unconditional transition to the next-state. An arrow may also omit an output.

For Drum, we will interpret Figure 8.1 as a bubble-and-arrow diagram for an FSM. Each of the yellow "bubbles" is a state, and the black arrows represent state-transitions. We will not annotate the arrows with labels, leaving the reader to refer to the BSV code.

8.2.1 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.[2]

Most non-trivial digital hardware systems are actually *concurrent FSMs*, *i.e.,* multiple classical FSMs running concurrently and independently and interacting with each other. Different BSV module instances are separate FSMs, each running their own process(es). These

[1] If the conditions are not mutually exclusive, we have a so-called *non-deterministic* FSM, where the next-state is chosen non-deterministically from all true conditions on outgoing arrows. In hardware design and in this book we are only concerned with deterministic FSMs, where the outgoing conditions will be mutually exclusive and we always have a deterministic, unique next-state.

[2] Even in non-deterministic FSMs, though there may be several possible next-states, exactly one next-state is non-deterministically chosen.

separate FSMs may communicate with each other *via* shared state (registers, fifos, register files, *etc.*).

For Fife, we will interpret Figure 8.1 as a set of concurrent FSMs. Each of the yellow boxes in the figure will be a separate module containing zero or more concurrent FSMs, and the black arrows represent communication between FSMs.

8.3 BSV: `StmtFSM`

The central BSV construct for temporal behavior (processes) is the "`rule`". Collections of rules can express any sequential or concurrent FSM. However, because Drum is a simple, structured sequential process, we can use a simpler, higher-level BSV notation called "`StmtFSM`" (which, in turn, is just converted into rules by the *bsc* compiler).

`StmtFSM` is a sub-language within BSV which is suitable for expressing *structured* sequential processes.[3] By "structured" we mean that processes are constructed by composing construct similar to those in most common sequential programming languages: blocks to express temporally sequenced actions, if-then-elses, while-loops and for-loops.

NOTE: We already briefly encountered a simple `StmtFSM`, with just a sequential block, in the testbench in Section 4.5.

8.3.1 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_F_to_D.deq;                 // Dequeue a fifo
3   f_D_to_RR.enq (v);            // Enqueue into a fifo
4   $display ("Hello, World!");   // Print something (in simulation only)
```

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`.

[3] `StmtFSM` can also express structured fork-join concurrency, but we do not need that capability for Drum.

8.3.2 `Action` blocks: grouping 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, there is no temporal ordering of these actions. All the primitive 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.

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).

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.

8.3.3 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.

The testbench in Section 4.5 contains an example of a `seq` block:

```
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.

8.3.4 StmtFSM: if-then-elses

Conditional 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 4.9 we described ordinary BSV if-then-else expressions. `StmtFSM` uses the same notation, but there is no ambiguity—the context always clearly distinguishes what we mean, because there is no overlap between ordinary expressions and `StmtFSM` constructions.

8.3.5 StmtFSM: while-loops

Repetitive processes can be expressed with traditional while-loop notation:

```
1  while (... Bool expression ...)
2     ... expression of type Stmt ...
```

8.3.6 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, not this particular `StmtFSM`.

NOTE: `StmtFSM` also has for-loop and repeat-loop notation, but we do not need them for Drum.

8.3.7 `StmtFSM`: `mkAutoFSM`: a simple FSM module constructor

Given an entity of type `Stmt`, we can pass it as an argument to to the module constructor `mkAutoFSM`

```
1    mkAutoFSM (... 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.

8.4 RISC-V: The interface for the Drum and Fife CPU modules

Armed with `StmtFSM` we can now complete our description of the Drum RISC-V CPU, a BSV module. Before we look at the module, we start with its interface. A clean, common interface will allows us later transparently to substitute the Fife CPU module in place of the Drum CPU module, and re-use the same test-benches *etc.*

(Please re-read Section 5.3.1 for the discussion on Harvard architectures, which have separate data channels for memory-access for instructions (Fetch, IMem) and memory-access for data (LOAD/STORE, DMem)).

```
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).

8.5 RISC-V: The Drum CPU module

Here is the Drum CPU module, except for the BEHAVIOUR section, which we have elided. We will fill in the missing piece (a `StmtFSM`) in a following subsection.

```
1  (* synthesize *)
2  module mkCPU (CPU_IFC);
3     // ================================================================
4     // STATE
5
6     // Don't run until initialized
7     Reg #(Bool) rg_running <- mkReg (False);
8
9     // The integer register file
10    RISCV_RegFile_IFC  gprs <- mkRISCV_RegFile;
11
12    // The Program Counter
13    Reg #(Bit #(XLEN)) rg_pc   <- mkReg (0);
14
15    // Inter-step registers
16    Reg #(F_to_D)             rg_F_to_D            <- mkRegU;
17    Reg #(D_to_RR)            rg_D_to_RR           <- mkRegU;
18    Reg #(RR_to_Retire)       rg_RR_to_Retire      <- mkRegU;
19
20    Reg #(RR_to_Control)      rg_RR_to_Control     <- mkRegU;
21    Reg #(Control_to_Retire)  rg_Control_to_Retire <- mkRegU;
22
23    Reg #(RR_to_EX)           rg_RR_to_EX          <- mkRegU;
24    Reg #(EX_to_Retire)       rg_EX_to_Retire      <- mkRegU;
25
26    // Paths to and from memory
27    FIFOF #(Mem_Req) f_IMem_req  <- mkFIFOF;
28    FIFOF #(Mem_Rsp) f_IMem_rsp  <- mkFIFOF;
29
30    FIFOF #(Mem_Req) f_DMem_req  <- mkFIFOF;
31    FIFOF #(Mem_Rsp) f_DMem_rsp  <- mkFIFOF;
32
```

```
33    // ================================================================
34    // BEHAVIOR
35
36    ... // This section will code the dynamic "behavior" of the module
37    ... // and will be discussed shortly
38
39    // ================================================================
40    // INTERFACE
41
42    method Action init (Initial_Params initial_params);
43       rg_pc      <= initial_params.pc_reset_value;
44       rg_running <= True;
45    endmethod
46
47    interface fo_IMem_req = to_FIFOF_O (f_IMem_req);
48    interface fi_IMem_rsp = to_FIFOF_I (f_IMem_rsp);
49
50    interface fo_DMem_NS_req = to_FIFOF_O (f_DMem_req);
51    interface fi_DMem_NS_rsp = to_FIFOF_I (f_DMem_rsp);
52 endmodule
53
```

In line 7 we instantiate a register that signals to the BEHAVIOR section when everything has been initialized and we are ready to start running.

In line 10 we instantiate the register file using the module `mkRISCV_RegFile` that was shown in Section 7.4.3. In line 13 we instantiates a register that will serve as our Program Counter.

Lines 12-21 instantiate registers that will hold values between temporal steps of the FSM. For example, the Fetch step will write a value into `rg_F_to_D` which will be read later by the Decode step. Not all registers will always contain meaningful values, but that's OK, each step will read only read from relevant registers at relevant points in time.

Lines 24-28 contain four FIFOs for IMem requests (outgoing) and responses (incoming) and DMem requests (outgoing) and responses (incoming) respectively. 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.

In the INTERFACE section, the `init` method lines 42-45 initializes the PC, and sets `rg_running` to true, releasing the BEHAVIOR section to start executing.

In lines 39-43 we are using the interface transformers discussed in Section 7.5.5 that produce Semi-FIFO "views" of FIFOs.

8.5.1 The Drum CPU module behavior

Here we fill in the BEHAVIOR section that was elided in the previous display. First we defines a `Stmt` that specifies the execution of a single instruction (Fetch through Retire):

```
1    Stmt exec_one_instr =
2    seq
```

```
3        // Fetch
4        action
5           let y <- fn_F (rg_inum, rg_pc, 0, 0);
6           rg_F_to_D <= y.to_D;
7           f_IMem_req.enq (y.mem_req);
8           rg_inum <= rg_inum + 1;
9        endaction
10       // Decode
11       action
12          let mem_rsp <- pop_o (to_FIFOF_O (f_IMem_rsp));
13          let y       <- fn_D (rg_F_to_D, mem_rsp);
14          rg_D_to_RR <= y;
15       endaction
16       // Register-Read and Dispatch
17       action
18          // Read GPRs
19          // Ok that read_1 and read_2 may return junk values
20          //         since not all instrs have rs1/rs2.
21          let x       = rg_D_to_RR;
22          let rs1_val = gprs.read_1 (instr_rs1 (x.instr));
23          let rs2_val = gprs.read_2 (instr_rs2 (x.instr));
24
25          Result_Dispatch y <- fn_Dispatch (rg_flog, x, rs1_val, rs2_val);
26
27          rg_RR_to_Retire  <= y.to_Retire;
28          rg_RR_to_Control <= y.to_Control;
29          rg_RR_to_EX      <= y.to_EX;
30       endaction
31       // Dispatch to one of the "execute" steps
32       if (rg_RR_to_Retire.exec_tag == EXEC_TAG_RETIRE)
33          // No-op
34          action
35          endaction
36       else if (rg_RR_to_Retire.exec_tag == EXEC_TAG_CONTROL)
37          // Control
38          action
39             let y <- fn_Control (rg_flog, rg_RR_to_Control);
40             rg_Control_to_Retire <= y;
41          endaction
42       else if (rg_RR_to_Retire.exec_tag == EXEC_TAG_IALU)
43          // IALU
44          action
45             let y <- fn_EX_IALU (rg_flog, rg_RR_to_EX);
46             rg_EX_to_Retire <= y;
47          endaction
48       else if (rg_RR_to_Retire.exec_tag == EXEC_TAG_DMEM)
49          // DMem
50          seq
51             action
52                Mem_Req y <- fn_DMem_Req (rg_flog, rg_RR_to_EX);
53                f_DMem_req.enq (y);
54             endaction
55             action
56                let mem_rsp <- pop_o (to_FIFOF_O (f_DMem_rsp));
```

```
57                let y       <- fn_DMem_Rsp (rg_flog, mem_rsp);
58                rg_EX_to_Retire <= y;
59             endaction
60          endseq
61       else
62          action
63             // This should be impossible
64             $finish (1);
65          endaction
66       // Retire
67       action
68          let y <- fn_Retire (rg_flog,
69                              rg_RR_to_Retire,
70                              rg_Control_to_Retire,
71                              rg_EX_to_Retire);
72          if (y.exception) begin
73             // Exception-handling not yet implemented
74             $finish (1);
75          end
76          else begin
77             // Update PC
78             rg_pc   <= y.to_F.next_pc;
79             // Update Rd if has rd-result
80             if (y.to_RW.commit)
81                gprs.write (y.to_RW.rd, y.to_RW.data);
82          end
83       endaction
84    endseq;
```

The code is very easy to read, in fact not too different from reading C/C++ code. The outer-level corresponds exactly to a left-to-right reading of Figure 8.1:

- an action for Fetch;
- an action for Decode;
- an action for Register-read-and-dispatch
- a nested if-then else performing one of the dispatched flows:
  - a no-op action (in case of direct information from RR to Retire), or
  - an action for Control; or
  - an action for IALU; or
  - a nested `seq` sequence of actions for DMem
    - an action to send a DMem request to memory;
    - an action to receive a DMem response from memory
- an action for Retire

Then, we instantiate a `StmtFSM` module that first waits until it is allowed to run, and then loops forever, executing one instruction at a time:

```
1  import StmtFSM :: *;
2
3  (* synthesize *)
4  module mkCPU (CPU_IFC);
5
```

```
6     ...
7
8     // ================================================================
9     // BEHAVIOR
10
11    Stmt exec_one_instr = ...;
12
13    mkAutoFSM (seq
14                   await (rg_running);
15                   while (True) exec_one_instr;
16               endseq);
17
18    // ================================================================
19
20    ...
21 endmodule
```

Exercise 8.1:

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 ...

□

8.6 RISC-V: Comparing Drum BSV CPU to C code for a RISC-V simulator

... TO BE WRITTEN ...

Chapter 9

RISC-V: Drum final topics: traps and interrupts

NOTE:

This chapter is to be written. The principal topics are:

- Extending RV32I with just enough CSRs and mechanisms to handle traps (for illegal instructions and memory faults)
- Extending RV32I to handle timer and external interrupts, so it can support an RTOS and perform basic performance measurement.

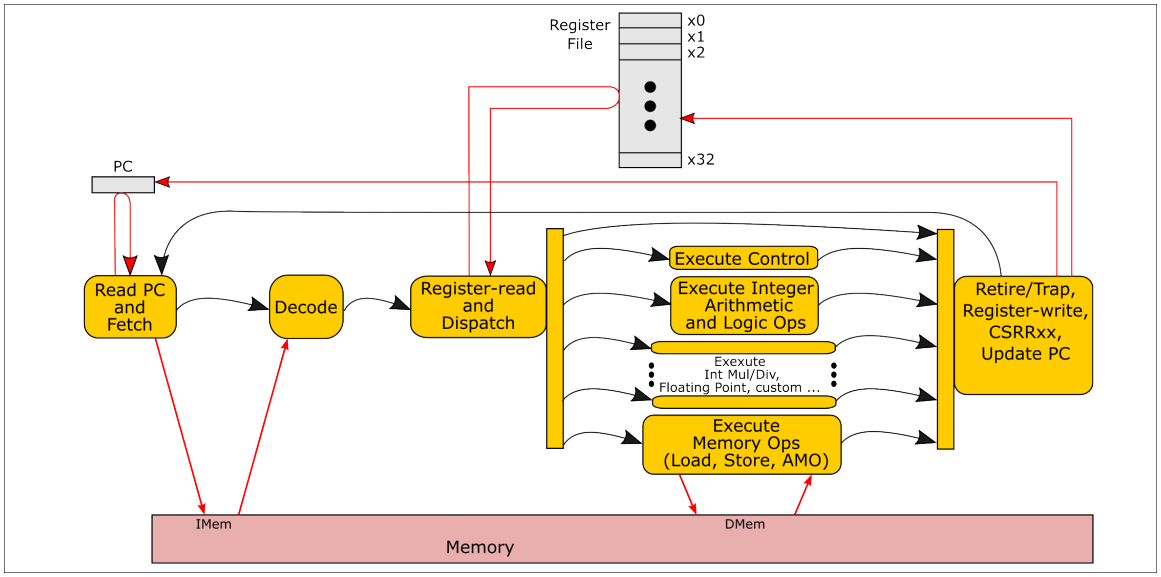

Figure 9.1: Simple interpretation of RISC-V instructions (same as Fig. 3.1)

9.1 Drum: Traps

Basic overview of how RISC-V does trap-handling is already covered in Chapter 2.

Here:

- Adding trap-handling CSRs (Control and Status Registers) MCAUSE, MTVAL, MEPC, MTVEC, MCAUSE, MRET, minimal MSTATUS.
- Adding CSRRxx instructions to access CSRs
- Adding trap-handling

9.2 Drum: Interrupts

Interrupts in Drum:

- 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. CSRs TIME, MCYCLE
- Interrupts are handled just like traps; the only question is: when to check for interrupts and respond.
- How does MIE bit return to 0?

9.3 CSRs for Performance Analysis

CSRs MTIME, MCYCLE, MINSTRET.

Mention "hpmcounter" CSRs for other events.

Chapter 10

RISC-V: the Fife pipelined CPU

10.1 Introduction

In this chapter we turn our attention to Fife, the pipelined CPU. We will reuse all the code already developed for Drum that implements basic RISC-V semantics. Our focus here is purely on pipelining and solving the new problems raised by pipelining.

Figure 10.1 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

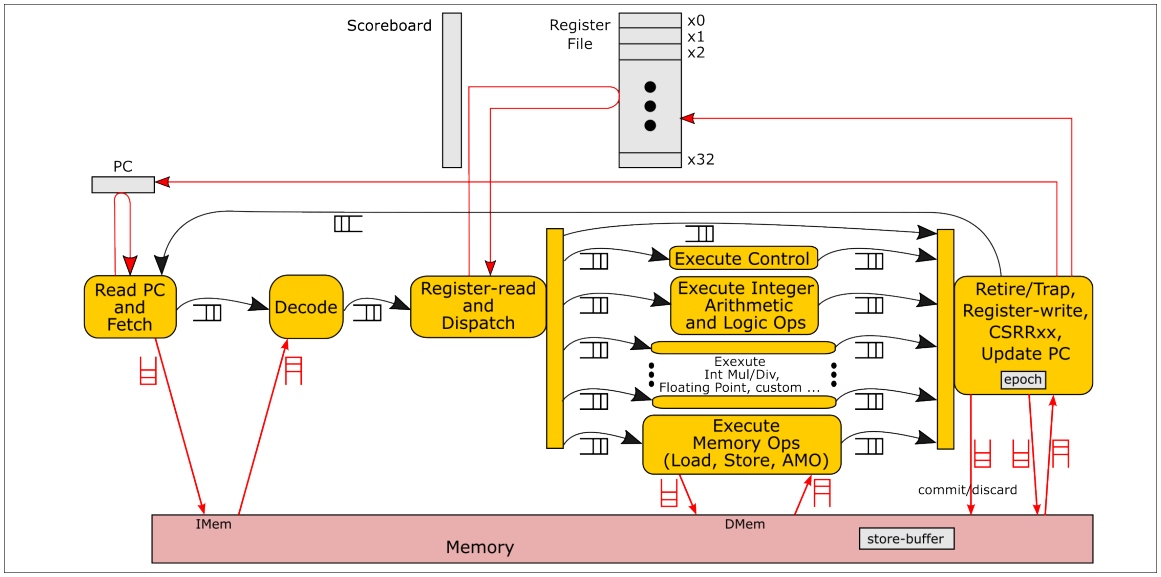

Figure 10.1: Pipelined interpretation of RISC-V instructions (Fig. 3.1 with new annotations)

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

10.2 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).

10.2.1 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.

10.2.2 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 10.1.

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 10.1:

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 10.2:

If `rg_epoch` contains a `Bit #(n)` value, how small can n be?

□

10.2.3 Mispredicted 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 mispredicted instructions make no modifications that are visible to right-path instructions that follow. The details of how this is accomplished will be seen in Section 10.5 and Section 10.6.

10.3 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 10.1.

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 10.3:

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 10.4:

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 10.5:

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 10.4.

□

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 10.6:

Implement a scoreboard unit with bypassing/short-circuiting as described in the above note.

Hint: Needs BSV "Concurrent Registers" (advanced topic!)

Exercise 10.7:

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.

10.4 Retiring outputs of the Execute Stages in Order with Tags

In Figure 10.1, 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 outpus 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 10.1 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 on the direct path to Retire indicating this, *i.e.,* 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. For example, if Retire sees a 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.

10.5 Allowing Memory Ops to be Pipelined with a Store Buffer

Instructions may attempt to write to memory. The `store buffer` holds these updates until we can deterimine whether the instruction is a wrong-path instruction or not. If it's a wrong path instruction, we can discard the update, else commit it to memory.

MMIO accesses may not be "memory-like". A LOAD can have a side effect; a STORE may have side-effects more than just writing a value, such as starting a motor or launching a rocket! Finally, a STORE to address A followed by a LOAD from address A may not return the same value. For such accesses, the "execute DMem" stage must defer the access until later when we are sure it is not a wrong-path instruction.

"Execute Memory Ops" stage is always speculative (because of mispredictions, and because older instructions in other pipes may trap).

- Need for store-buffer and final commit/discard from Retire-stage
- Mem-system performs store only in store-buffer
- Retire stage sends commit/discard message to store-buffer. Discuss: do we need 'discard' messages, or are 'commit' messages enough?

What about MMIO?

- LOADs may have side-effects
- LOADs may not be idempotent
- STOREs may have side-effects in addition to value stored
- LOAD may not return most-recently stored value

So, these cannot be done speculatively, *i.e.,* in "Execute Memory Ops" stage.

- 'Execute Memory ops' stage defers the request by sending it back as another kind of 'response'
- Retire step performs it once we know it is non-speculative.

10.6 The Retire Stage

10.7 Fife: CSRs

- CSRRxx are read-modify-write operations

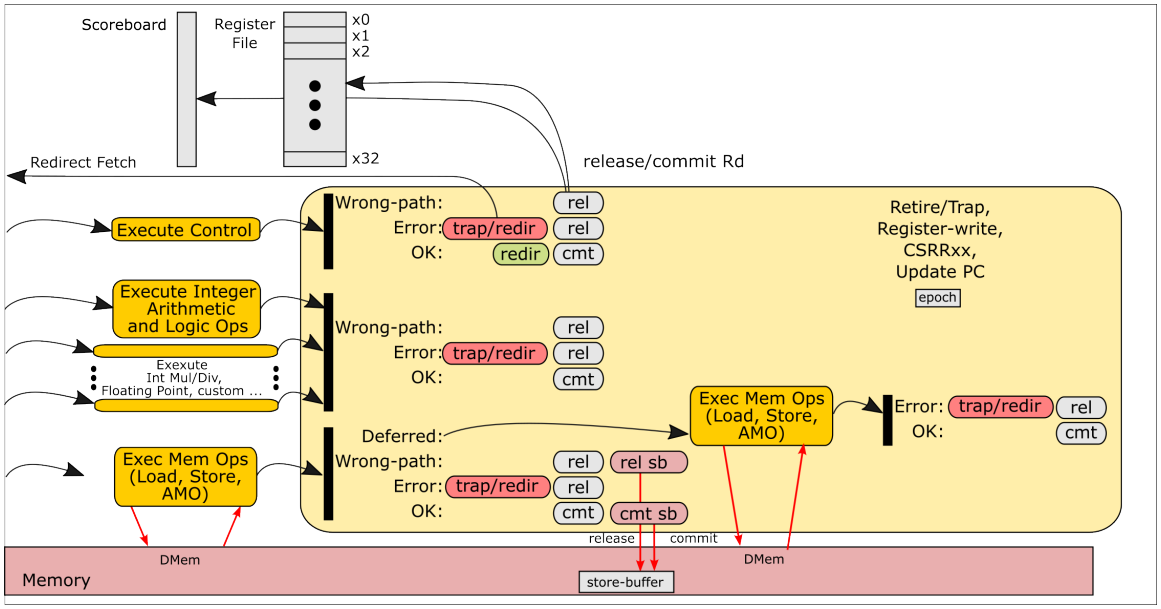

Figure 10.2: Actions in the "Retire" stage of Fife

- CSRRxx access may not be memory-like (side-effecting reads, read may not return last written value,
- ... (a bit like MMIO issues)

Hard to pipeline, so execute in Retire stage, as FSM.

CSRRxx instructions should be rare, so FSM exec does not affect overall performance.

10.8 Fife: Interrupts

Sample for interrupts in Retire stage, fix up CSRs and and redirect.

Retire stage already has infra for CSR update and redirection, so this is a small incremental change.

Chapter 11

Pending (to be written): Advanced topics, possibly in "Book 2"?

This is a place-holder chapter with a TODO list of pending possible topics to be written.

Goal: Linux capability, better performance.

11.1 Advanced branch prediction

Is a form of online machine-learning.

Branch instruction hints.

Branch-Target buffers (BTBs)

Return-Address Stacks (RASs)

Hysteresis in prediction.

11.2 Caches

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.

Cache-coherence.

Non-blocking caches.

11.3 Compressed instructions

Implementation: wholly in Decode stage. Common 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.

11.4 AMO operations

Note: These (implementation-independent) explanations may already be done in Chapter 2 on the RISC-V ISA:

- Explanation of LR and SC instructions
- Explanation of AMOxxx instructions
- Implementation in the memory system

11.5 Multiple Privilege levels

Machine, Supervisor, User

Interrupt/trap delegation.

Hypervisor

11.6 Memory Protection with PMPs

Page Tables, TLBs

11.7 Virtual Memory

Page Tables, TLBs

11.7.1 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 [14] contains a link to `riscv.org` from which to download a PDF.
- "The RISC-V Instruction Set Manual Volume II: Privileged Architecture"
  Bibliography entry [15] 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^{TM} simulator; they can be synthesized to Verilog and simulated on a Verilog simu-

lator; 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 programming 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.

[1] SystemVerilog and SystemC have object-orientation, polymorphism, and overloading, but these are typically used only in simulation for verification environments of hardware designs, not for actual hardware design itself.

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.

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

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.

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, ...

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).

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

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] 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/.

[15] 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/.