# Designing a Simple Pipelined RISC-V CPU
(using BSV)

*or*

# Learning BSV for Digital Design
(using a Simple Pipelined RISC-V CPU)

Rishiyur S. Nikhil

Bluespec, Inc.




DRAFT: December 26, 2023, 15:40h 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

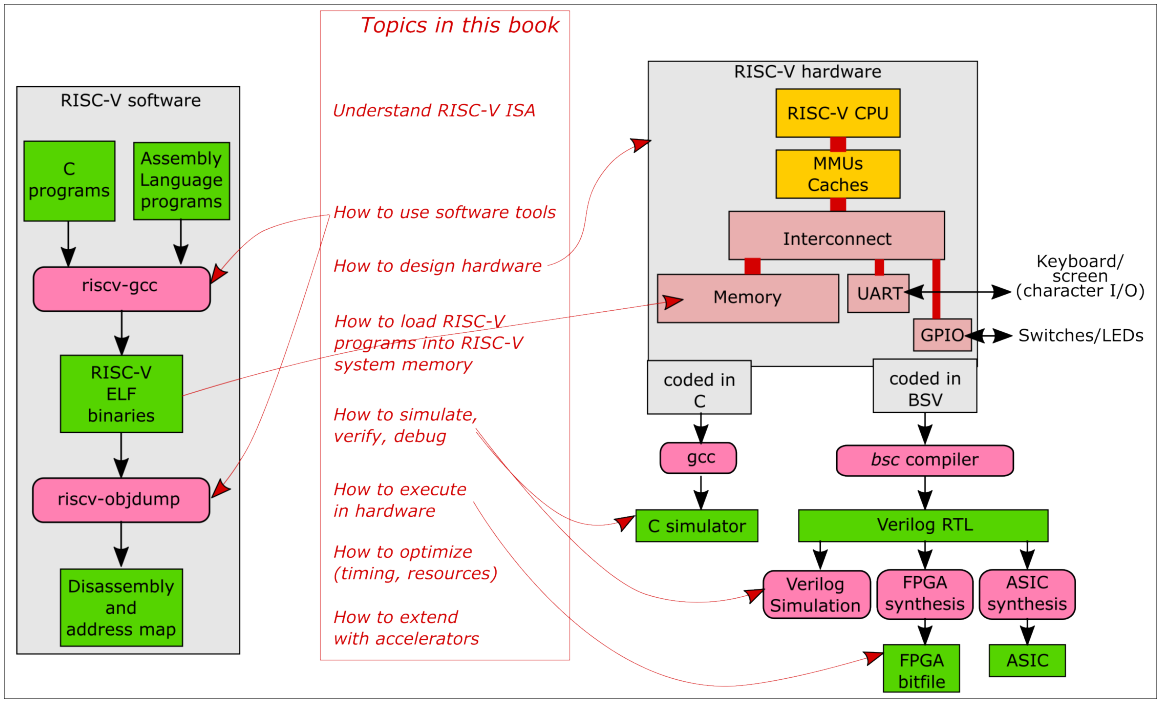


Figure 1.1: Topics covered (in red text in red box)

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 "Magritte", a *non-pipelined* implementation[1] 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 Magritte. 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.

[1] See René Magritte, Belgian surrealist painter and his famous piece labeled *"Ceci n'est pas une pipe"* ("This is not a pipe") https://en.wikipedia.org/wiki/Rene_Magritte

# 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.

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—"Magritte"—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).

Magritte 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).

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 Magritte. In fact, we will reuse the functional code from Magritte without change.

For both Magritte 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 Magritte 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, Magritte 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.

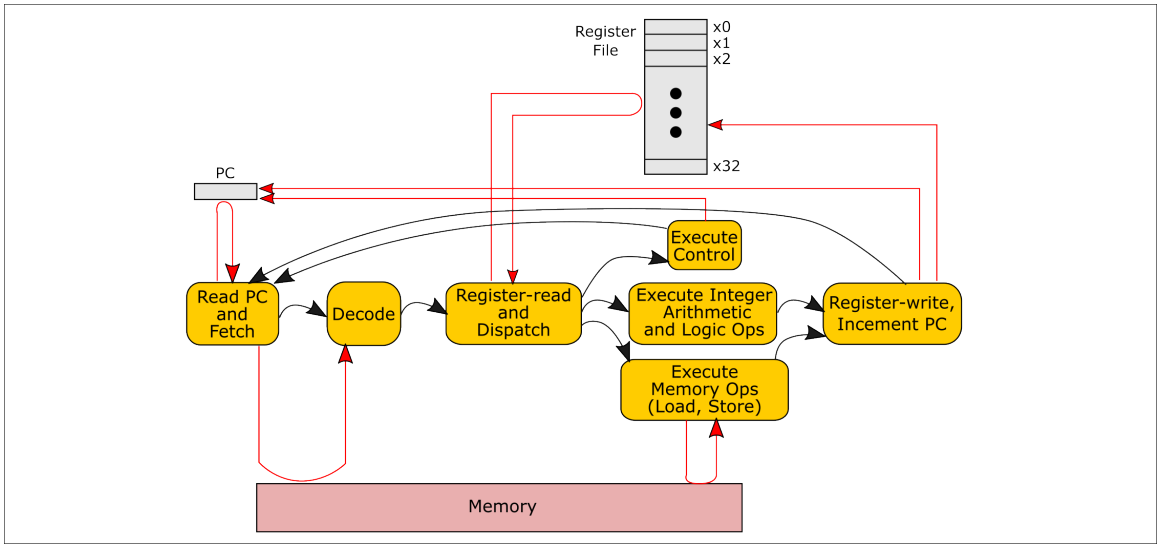


Figure 3.1: Simple interpretation of RISC-V instructions

- 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 Tactics: the order in which we tackle the 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"). To serve the BSV-learning purpose, we will tackle the algorithm steps of Figure 3.1 in a different order from the logical order of the diagram.

We will start, in the next chapter, with the Decode step because, in a certain hardware sense, it is conceptually the simplest. In later chapters we will move on to the Fetch step and then the remaining steps.

# Chapter 4

# BSV Combinational circuits for the RISC-V Decode step

## 4.1 Introduction

In this chapter we focus on learning enough BSV to write code for the Decode step of the RISC-V interpretation steps shown in Figure 3.1. As mentioned earlier, we choose to start with the Decode step (instead of the Fetch step) to facilitate the BSV-learning process.

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

- A 32-bit piece of data—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:

- Is it a legal 32-bit instruction?
- If legal, what is its broad classification: Branch? Integer Arithmetic or Logic? Memory Access? This will help the next step in dispatching to one of the Execute steps.
- 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. We need to bundle together all these bits of output information so that we can pass to to the next step, for which we need some kind of "struct" or "record" structure. Finally, as in all good programming language methodology, we'd like to package up all this functionality inside a "function" with clearly specified input(s) and output(s).

[1] In RISC-V, so-called "compressed" instructions can be 16-bits, but we ignore that for now

In the next several sections—4.2, 4.3, 4.9, 4.4—we will learn the BSV concepts needed to code the Decode step. Then, in Section 4.12 we use these concepts to show the BSV code for the Decode function.

## 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:

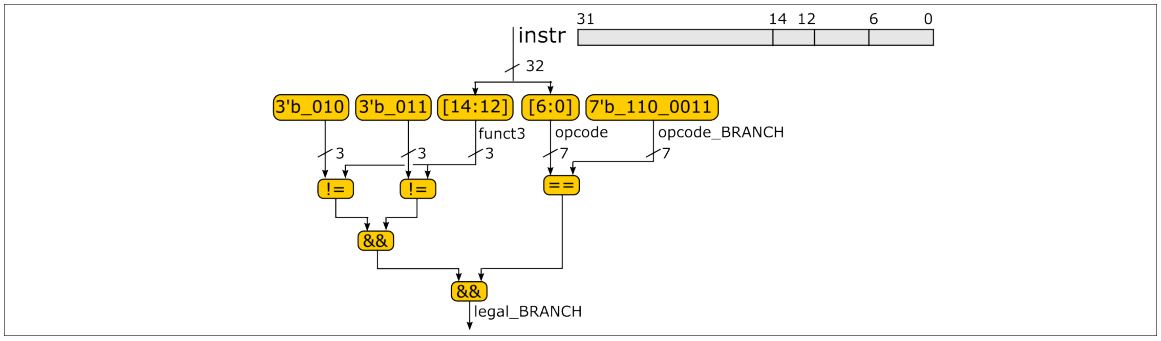


Figure 4.2: Testing for a legal BRANCH instruction

- 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 6.3). 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
```

### 4.5.1 Exercises

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

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` ...

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).

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).

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 (`OPCODE_ILLEGAL`, `OPCODE_CONTROL`, ...).

## 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 Bool 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 Bool instr_opclass (Bit #(32) instr);
   return ((is_legal_BRANCH (instr)
            || is_legal_JAL (instr)
            || is_legal_JALR (instr))
           ? OPCLASS_CONTROL
           : OPCLASS_ILLEGAL);
endfunction
```

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

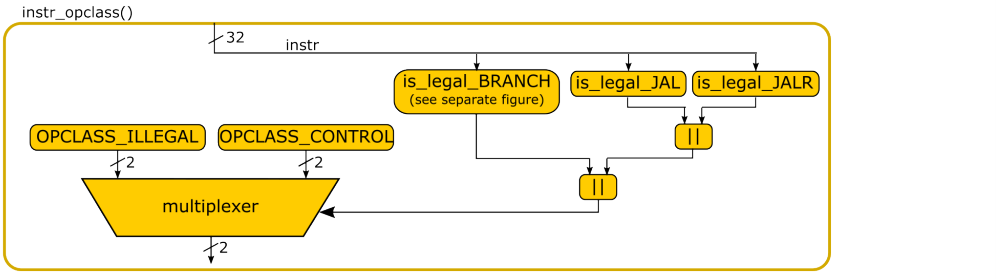


Figure 4.3: If-then-else is a multiplexer

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

If-then-elses can of course be nested:

```
function Bool instr_opclass (Bit #(32) instr);
   OpClass result;
   if (is_legal_BRANCH (instr)
      || is_legal_JAL (instr)
      || is_legal_JALR (instr))
      result = OPCLASS_CONTROL;
   else if (is_legal_IALU (instr))
      result = OPCLASS_IALU;
   else if (is_legal_Mem (instr))
      result = OPCLASS_MEM;
   else
      result = OPCLASS_ILLEGAL;
   return result;
endfunction
```

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

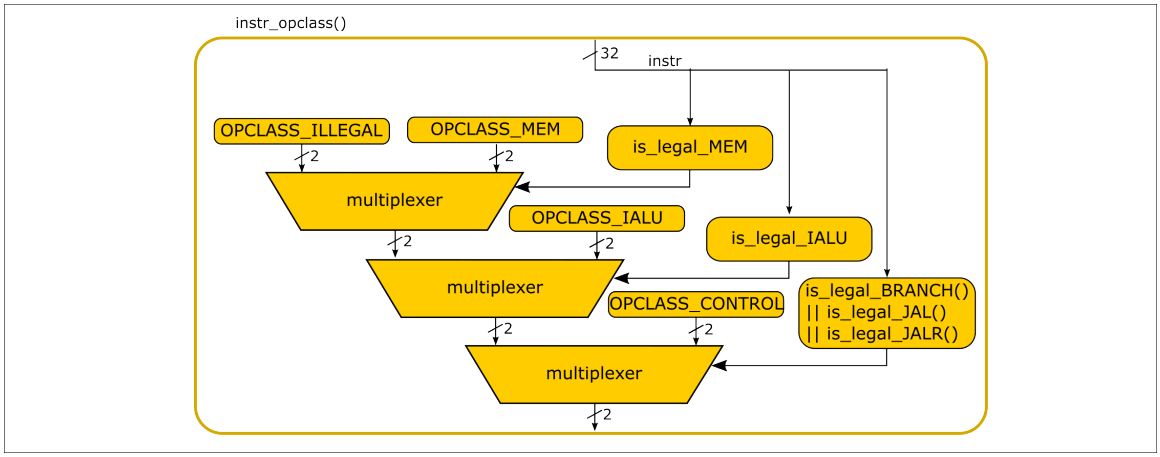


Figure 4.4: Nested if-then-elses become cascaded multiplexers

### 4.9.1 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.

### 4.9.2 Exercises

1. 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()`.

2. 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.

3. 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.

4. 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]

### 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
```

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

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/Magritte's `Arch.bsv` file:

```
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.

## 4.11 BSV: `struct` types

In the Decode step of Figure 3.1 we want to produce several results that will be passed on to subsequent steps. These include:

- The PC. This will be needed by BRANCH and AUIPC instructions to compute addresses that are offset from the current PC. It will be needed in JAL instructions which save PC+4 or PC+2 as a return-address. It will be needed for any traps (exceptions) that may occur, which save PC for the trap-handler.
- The instruction, itself. This will be needed for opcode details, the rs1, rs2 and rd register indexes, immediate values, *etc.*
- The `OpClass`. This will be needed to dispatch the instruction to the appropriate handling step.
- 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, which is formatted differently depending on the class of instruction.

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

```
typedef struct {Bit #(64)   inum;    // for debugging only
                Bit #(XLEN) pc;
                Bit #(32)   instr;

                OpClass     opclass;
                Bool        has_rs1;
                Bool        has_rs2;
                Bool        has_rd;
                Bit #(32)   imm;
} D_to_RR
deriving (Bits, FShow);
```

Because we said "`deriving(Bits)`", the *bsc* compiler will automatically work out a representation for `DD_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. If we had not said "`deriving(Bits)`", we could explicitly provide some other custom representation in bits.

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 omething like this:

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

### 4.11.1 Creating struct values

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

```
     DD_to_RR x = D_to_RR {inum:     ... value of field ...,
                           pc:       ... value of field ...,
                           instr:    ... value of field ...
                           ...};
```

The repetition of `DD_to_RR` above seems verbose; the left-hand side instance is the type, and the right-hand side instance is the "struct constructor". 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`":

```
     let x = D_to_RR {inum:     ... value of field ...,
                      pc:       ... value of field ...,
                      instr:    ... value of field ...
                      ...};
```

### 4.11.2 Selecting struct fields

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

```
x.pc
x.instr
```

### 4.11.3 Updating struct fields using assignment

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

```
x.pc    = ... new value ...
x.instr = ... new value ...
```

## 4.12 RISC-V: The Decode function

We are now in a position to write Figure 3.1's Decode function:

```
function D_to_RR fn_D (Bit #(64) inum, Bit #(XLEN) instr, Bit #(32) instr);
   Bit #(5) rd = instr_rd (instr);

   // Baseline info to next stage
   let d_to_rr = D_to_RR {inum:     inum,
                          pc:       pc,
                          instr:    instr,

                          opclass:  OPCLASS_ILLEGAL,
                          has_rs1:  False,
                          has_rs2:  False,
                          has_rd:   False,
                          imm:      0};

   if (is_legal_LUI (instr) || is_legal_AUIPC (instr)) begin
      d_to_rr.opclass = OPCLASS_IALU;
      d_to_rr.has_rd  = (rd != 0);
      d_to_rr.imm     = zeroExtend (instr_imm_U (instr));
   end

   else if (is_legal_BRANCH (instr)) begin
      d_to_rr.opclass = OPCLASS_CONTROL;
      d_to_rr.has_rs1 = True;
      d_to_rr.has_rs2 = True;
      d_to_rr.imm     = zeroExtend (instr_imm_B (instr));
   end

   else if (is_legal_JAL (instr)) begin
```

```
29          d_to_rr.opclass = OPCLASS_CONTROL;
30          d_to_rr.has_rd  = (rd != 0);
31          d_to_rr.imm     = zeroExtend (instr_imm_J (instr));
32       end
33
34       else if (is_legal_JALR (instr)) begin
35          d_to_rr.opclass = OPCLASS_CONTROL;
36          d_to_rr.has_rs1 = True;
37          d_to_rr.has_rd  = (rd != 0);
38          d_to_rr.imm     = zeroExtend (instr_imm_I (instr));
39       end
40
41       else if (is_legal_LOAD (instr)) begin
42          d_to_rr.opclass = OPCLASS_MEM;
43          d_to_rr.has_rs1 = True;
44          d_to_rr.has_rd  = (rd != 0);
45          d_to_rr.imm     = zeroExtend (instr_imm_I (instr));
46       end
47
48       else if (is_legal_STORE (instr)) begin
49          d_to_rr.opclass = OPCLASS_MEM;
50          d_to_rr.has_rs1 = True;
51          d_to_rr.has_rs2 = True;
52          d_to_rr.imm     = zeroExtend (instr_imm_S (instr));
53       end
54
55       else if (is_legal_OP_IMM (instr)) begin
56          d_to_rr.opclass = OPCLASS_IALU;
57          d_to_rr.has_rs1 = True;
58          d_to_rr.has_rd  = (rd != 0);
59       end
60
61       else if (is_legal_OP (instr)) begin
62          d_to_rr.opclass = OPCLASS_IALU;
63          d_to_rr.has_rs1 = True;
64          d_to_rr.has_rs2 = True;
65          d_to_rr.has_rd  = (rd != 0);
66       end
67
68       return d_to_rr;
69    endfunction
```

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. 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.

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.

### 4.12.1 Exercises

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

2. Write a testbench for `fn_D()`, apply it on a number of 32-bit values (instructions) and print the results.

# Chapter 5

# The Fetch function: memory requests and responses

## 5.1 Introduction

In this chapter we focus on learning enough BSV to code the front part of the algorithm shown in Figure 3.1, which we repeat here for convenience. We have already seen the Decode

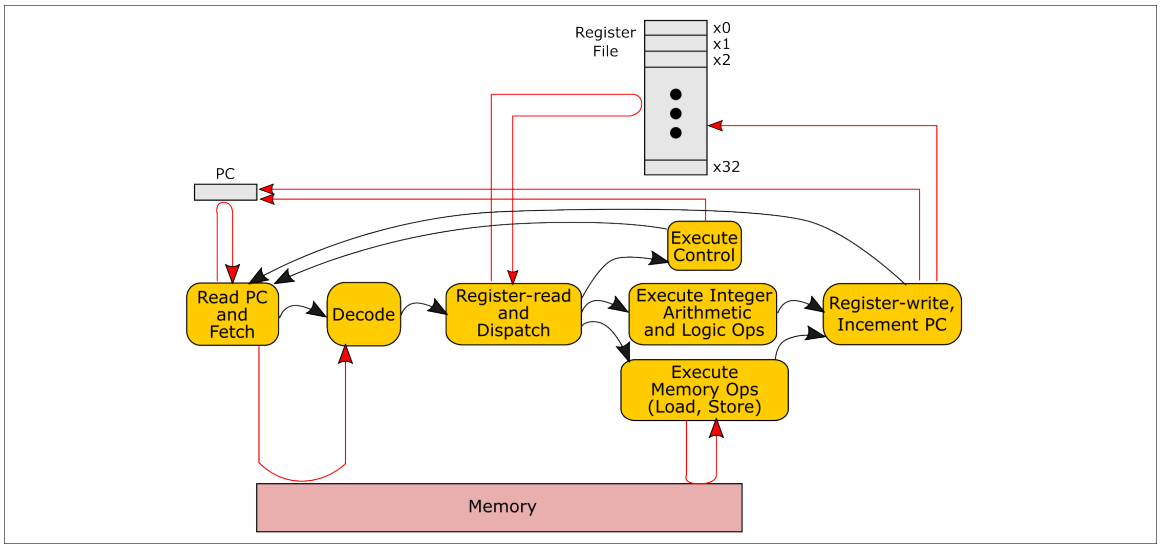


Figure 5.1: Simple interpretation of RISC-V instructions (same as Fig. 3.1)

function in the previous chapter. Now we will add the Read-PC-and-Fetch functionality, and then place this, along with the Decode function, into a Finite State Machine that will continually fetch and decode instructions.

The outputs of Read-PC-and-Fetch step are:

- A *memory request* to memory, to read an instruction.
- Some additional information passed on to the Decode step for subsequent use.

## 5.2 RISC-V: Memory Requests

A memory-request is either for reading data ("load") or for writing data ("store"). We express these request-types using an enum type:

```
typedef enum {MEM_LOAD,
              MEM_STORE} Mem_Req_Type
deriving (Eq, FShow, Bits);
```

A memory-request in RISC-V RV32I may be for one, two or four bytes. We express these request-size options using an enum type:

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

Finally, 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:

```
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, 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 RISC-V: Address Alignment

Although nowadays we think of all computer memories in units of bytes, and being byte-addressed, 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]`).

We can see why address-alignment is desirable. In a memory organized according to wider words, a misaligned read request may straddle word boundaries and so may require reading two words. In a memory system with caches, a misaligned read request may straddle a cache-line boundary, and may require two accesses, which may hit or miss independently. In a memory system with virtual memory, a misaligned read request may straddle a page

boundary, and may require two accesses, which may hit or page-fault independently. In short, misaligned accesses add complexity to memory-system hardware design.

We can organize our software so that misaligned accesses are exceedingly rare. Since most software is produced by compilers, we can make the compiler ensure that instructions and data are placed in memory at aligned addresses, possibly by padding gaps between "adjacent" small data (such as two byte-sized fields in a struct).

Rather than paying the price of additional complexity to handle misaligned accesses, to be used only in rare cases when accesses are actually misaligned, memory-system designers usually choose instead to return an error on misaligned accesses. The error will cause the CPU to "trap", and special trap-handling code can then recover by explicitly performing two, smaller, aligned accesses.

## 5.4 RISC-V: Memory Responses

The response from memory for a 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 (*e.g.*,, although RV32I addresses are 32-bits, which can address 4GiB of memory, we may provision our system with something smaller, say 1 GiB);
- Corruption of data, as detected by parity bits or some other error-detection mechanism).

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

```
typedef enum {MEM_OK, MEM_MISALIGNED, MEM_ERR} Mem_Rsp_Type
deriving (Eq, FShow, Bits);
```

A memory-response contains the response-type and, for LOAD requests that with OK response-type, 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, we assume the data is passed in the least-significant bytes of the data field.

// ****************************************************************

## 5.5 RISC-V: more structs for the Fetch function

In Figure 5.1, the information passed from the Fetch stage to the Decode stage is just the PC:

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

It might seem like overkill to define a struct for just one field like this, but it has the following advantages:

- It becomes easy to add more fields later, should we need to do so. In particular for Fife will 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 irrelevant 32-bit value in place of an `F_to_D` struct.
- And `F_to_D` value occupies the same XLEN bits as the `pc` field by itself; there is no overhead just because we used a struct.

Structs can be nested. The Fetch function's result is a nested struct containing two structs, one being the memory-request to be sent to memory, and the other being the information to be passed on to the Decode step:

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

## 5.6 RISC-V: The Fetch Function

Finally, we are ready to write the Fetch function which is very simple. Its input is the current value of the program counter (PC), and it returns a `Result_F`. The PC is used as the address from which to read memory.

```
function Result_F fn_F (Bit #(XLEN)  pc)
    Result_F y = ?;

    // Info to next stage
    y.to_D = F_to_D {pc: pc};

    // Request to IMem
    y.mem_req = Mem_Req {req_type: MEM_LOAD,
                         size:     MEM_4B,
```

```
10                              addr:     pc,
11                              data :    0};
12        return y;
13  endfunction
```

# Chapter 6

# Magritte Fetch and Decode: Modules, Interfaces, Finite State Machines

## 6.1 BSV: Modules: state, behavior and interfaces

Any non-trivial design needs to be organized into smaller, manageable, modules that interact with each other. This is true for any programming language, and BSV is no exception. For Magritte, we will want to package the top-level system to contain the CPU and the memory; The CPU itself will contain a register-file module and the code for its behavior; and so on.

In this section we describe the concept of modules and interfaces in BSV. This section may require re-reading a couple of times; the concepts become properly internalized only after seeing/using/creating a couple of examples, which we will do in subsequent sections.

Modules and interfaces clearly separate the concerns of externally-visible functionality ("external API", *what* a module does) *vs.* internal implementation details (*how* the module does it). BSV's style of modules is very "object-oriented" in flavor: a module is like an object constructor; a module instance is like an object; its interface is a set of methods that can be invoked like functions.

An *Interface Declaration* in BSV declares a new interface 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 specifies a set of *methods* that can be invoked from an external context. Methods have arguments and results, and can change internal state in the module. In the `interface` declaration, we only describe a method's argument and result types, *i.e.,* its externally visible features.

Interfaces can be nested (can contain sub-interfaces).

An `interface` declaration just defines a new type. There can be any number of module declarations each of which offers the same interface. For example, the BSV library contains a repertoire of FIFO modules, all of which have the same FIFO interface type.

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

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

A `module` declaration declares a module constructor; it can be invoked multiple times to obtain multiple *instances*.

### 6.1.1 BSV: Registers

BSV treats all "state elements" (components that store persistent values) uniformly as modules with interfaces. This includes "primitives" like ordinary registers. Further, unlike Verilog and SystemVerilog, registers in BSV are subject to strong type-checking; each register only contains a value of a particular type.

#### BSV: Register interfaces

The standard register interface type in BSV is:

```
interface Reg #(t);
   method t _read();
   method Action _write (t x);
endinterface
```

Here, "`t`" is the type of value stored in the register. Registers are accessed using two methods. 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()`.

#### BSV: `mkReg`, a register module (consructor)

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

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

Here we are declaring a new identifier `pc` with type `Reg#(Bit#(XLEN))` (the register interface type) and binding 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.

### 6.1.2 BSV: FIFOs

FIFOs (First-in-First-out) elements are queues of values. 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. BSV FIFOs are normally "flow-controlled", *i.e.,* it is impossible to enqueue into a full FIFO, and to dequeue from an empty FIFO. FIFOs in BSV are subject to strong type-checking; each FIFO can only contain values of one particular type.

#### BSV: FIFOF interfaces

A standard FIFO interface type in the BSV library is:

```
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
```

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

The `first()` and `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 alter the state of the FIFO. The `deq()` method discards the value at head of the queue and advances the queue.

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

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

Note that all values in a particular FIFO have the same type `t`.

We will also find it useful to use the following interfaces for each "end" of a FIFO queue:

```
1 interface FIFOF_O #(t);
2    method Bool notEmpty();
3    method t first();
4    method Action deq();
5 endinterface
```

```
1 interface FIFOF_I #(t);
2    method Bool notFull();
3    method Action enq (t x);
4 endinterface
```

**BSV: `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 Magritte we use a standard FIFO module (constructor):

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

Here we declare a new identifier `f_to_IMem` with type `FIFOF#(Mem_Req)` (the FIFOF interface type, with the values in the queue having type `Mem_Req`) and bind it to the interface offered by a newly instantiated FIFO.

## 6.2 RISC-V: The interface of the CPU module for Magritte and Fife

```
1  interface CPU_IFC;
2     interface FIFOF_O #(Mem_Req) fo_IMem_req;
3     interface FIFOF_I #(Mem_Rsp) fi_IMem_rsp;
4
5     interface FIFOF_O #(Mem_Req) fo_DMem_req;
6     interface FIFOF_I #(Mem_Rsp) fi_DMem_rsp;
7  endinterface
```

## 6.3 BSV: Finite State Machines (FSMs)

So far, we have only been discussing pure combinational functions, for which there is no concept of time. Combinational functions are just "instantaneous" transformers from input values to output values. We are now ready to start discussing *processes*— behaviors that evolve over time. The general BSV construct for this is the "`rule`", but for Magritte we can use a higher-level BSV construct called "`StmtFSM`" (which are implemented by the *bsc* compiler in terms of rules.

## 6.4 RISC-V: A partial Magritte CPU with Fetch and Decode

```
1  (* synthesize *)
2  module mkCPU (CPU_IFC);
3     FIFOF #(Mem_Req) f_to_IMem   <- mkFIFOF;
4     FIFOF #(Mem_Rsp) f_from_IMem <- mkFIFOF;
5
6     // The integer register file
7     RegFile #(XLEN)    gprs  <- mkRegFileFull;
8
```

```
9     Reg #(Bit #(XLEN)) rg_pc <- mkReg (0);
10
11    // Inter-stage registers
12    Reg #(F_to_D)                      rg_F_to_D                   <- mkRegU;
13
14    // ================================================================
15
16    mkAutoFSM (
17       seq
18          // Fetch
19          action
20              let y <- fn_F (rg_pc);
21              rg_F_to_D <= y.to_D;
22              f_to_IMem.enq (y.mem_req);
23          endaction
24
25          // Decode
26          action
27             Mem_Rsp rsp_IMem = pop_o (to_FIFOF_O (f_from_IMem));
28             let y = fn_D (rg_F_to_D, rsp_IMem);
29             rg_D_to_RR <= y;
30          endaction
31       endseq);
32
33    // ================================================================
34
35    interface Client_Semi_FIFOF cl_IMem;
36       interface request  = to_FIFOF_O (f_to_IMem);
37       interface response = to_FIFOF_I (f_from_IMem);
38    endinterface
39 endmodule
40
```

## 6.5 RISC-V: Partial CPU with Fetch and Decode

## 6.6 Topics

- CPU state machine
- Top-level, connecting to memory

# 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.

**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/.