Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L8: RISC-V: GPRs and CSRs

Reminders

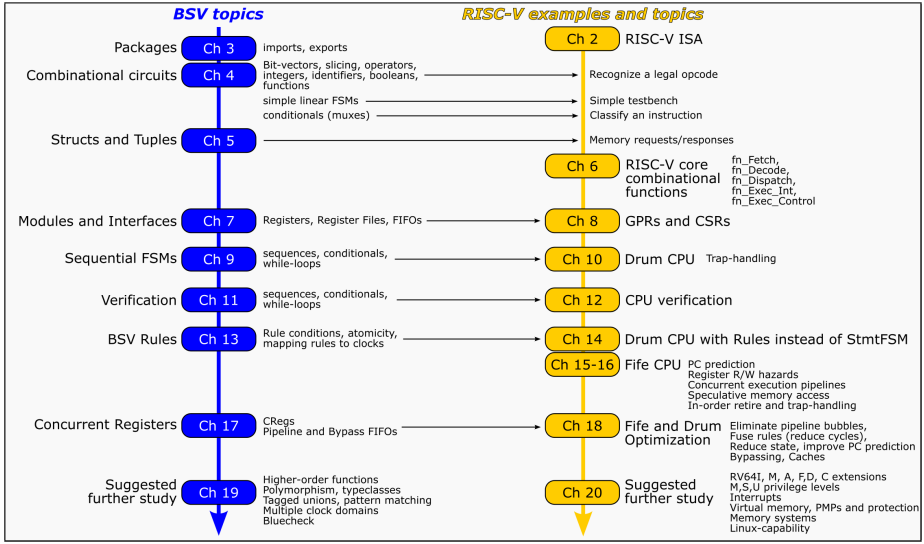
Please git clone or git pull: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design

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

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

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

Chapter Roadmap



Flow of information between stages in Drum and Fife

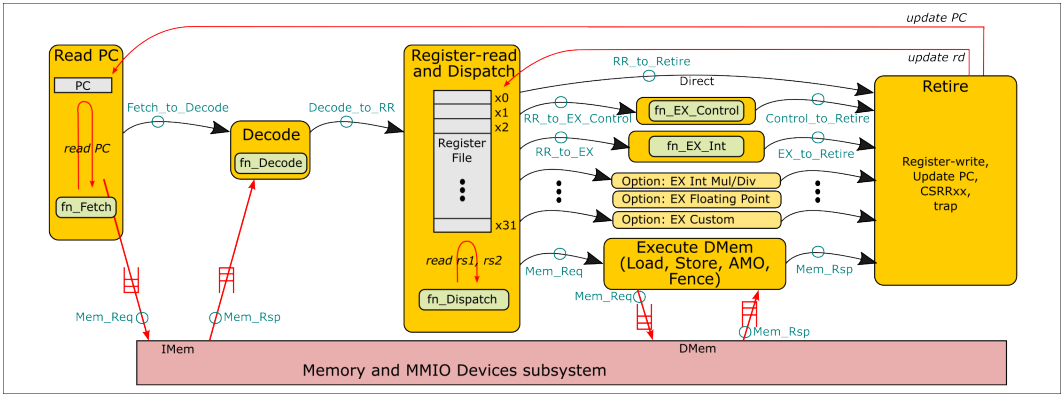


Table of Contents

GPRs (General Purpose Register file)

RISC-V: Interface for the mkGPRs module

Although we could have just used the **BSV** library "RegFile" interface, we define a new interface that specializes it to the particular types in our application, and with more meaningful names:

src_Common/GPRs.bsv: line 25 ...

```
interface GPRs_IFC #(numeric type xlen);
   method Bit #(xlen) read_rs1 (Bit #(5) rs1);
   method Bit #(xlen) read_rs2 (Bit #(5) rs2);
   method Action      write_rd (Bit #(5) rd, Bit #(xlen) rd_val);
endinterface
```

Note:

- This is a polymorphic module; "xlen" can be instantiated with the numeric type 32 for RV32 and 64 for RV64.
- Each method in a module interface can be invoked at most once in a clock. We will need to read rs1 and rs2 registers simultaneously, and so we provide a separate method for each.

RISCV: Register x0 is a special case

In RISC-V ISA semantics, register x0 (index 0) is defined as "always zero". Any value written to x0 is ignored/discarded, and any read from x0 always returns 0.

The `mkGPRs` module is a thin wrapper for the **BSV** library `mkRegFileFull` module, treating index 0 as a special case.

src_Common/GPRs.bsv: line 41 ...

```
module mkGPRs (GPRs_IFC #(xlen));
   RegFile #(Bit #(5), Bit #(xlen)) rf <- mkRegFileFull;

   method Bit #(xlen) read_rs1 (Bit #(5) rs1);
      return ((rs1 == 0) ? 0 : rf.sub (rs1));
   endmethod

   method Bit #(xlen) read_rs2 (Bit #(5) rs2);
      return ((rs2 == 0) ? 0 : rf.sub (rs2));
   endmethod

   method Action write_rd (Bit #(5) rd, Bit #(xlen) rd_val);
      rf.upd (rd, rd_val);
   endmethod
endmodule
```

Exercise break

Please see Appendix E, Exercise Ex-08-A-GPR-Register-Files.

RISC-V: CSR addresses

A CSR address is 12-bits wide (taken from `instr[31:20]` in CSRRxx instructions).

Here are the addresses for the CSRs we need for exception-handling:

src_Common/CSR_Bits.bsv: line 23 ...

```
Bit #(12) csr_addr_MTVEC     = 'h305;
Bit #(12) csr_addr_MEPC      = 'h341;
Bit #(12) csr_addr_MCAUSE    = 'h342;
Bit #(12) csr_addr_MTVAL     = 'h343;
```

RISC-V: Interface to the mkCSRs

The interface methods for mkCSRs reflects the way we use CSRs:

src_Common/CSRs.bsv: line 29 ...

```
interface CSRs_IFC;
   method Action init (Initial_Params initial_params);

   // CSRRXX instruction execution
   // Returns (True, ?) if exception else (False, rd_val)
   method ActionValue #(Tuple2 #(Bool, Bit #(XLEN)))
          mav_csrrxx (Bit #(32) instr, Bit #(XLEN) rs1_val);

   // Trap actions
   // Returns PC from MTVEC for trap handler
   method ActionValue #(Bit #(XLEN))
          mav_exception (Bit #(XLEN) epc,
                         Bool        is_interrupt,
                         Bit #(4)    cause,
                         Bit #(XLEN) tval);

   method Bit #(XLEN) read_epc;
   method Action ma_incr_instret;

   // Set TIME
   (* always_ready, always_enabled *)
   method Action set_TIME (Bit #(64) t);
endinterface
```

RISC-V: CSRs: General considerations

Technically, there can be 2^{12} (= 4096) CSRs.

In Drum/Fife, we implement hardly a dozen CSRs.

Even high-end CPUs may implement around hundred CSRs.

Further,

- The addresses of CSRs that we do implement are not consecutive.
- What happens when we read/write a CSR may vary widely for different CSRs and may have CSR-specific side effects. (See Sections 2.3, 2.4 and 2.6 in of the Privileged ISA Specification.)

Thus, we cannot use, say, the library `RegFile` module.

Instead, we implement CSRs using an *ad hoc* collection of separate registers, and *ad hoc* read/write logic for each CSR. Here is the code for the CSRs we need for exception-handling:

src_Common/CSRs.bsv: line 61 ...

```
Reg #(Bit #(XLEN)) csr_mtvec   <- mkReg (0);
Reg #(Bit #(XLEN)) csr_mepc    <- mkReg (0);
Reg #(Bit #(XLEN)) csr_mcause  <- mkReg (0);
Reg #(Bit #(XLEN)) csr_mtval   <- mkReg (0);
```

RISC-V: mkCSRs module

Please view the actual code in: `Code/src_Common/CSRs.bsv`

Start by viewing the internal functions
`fn_fav_csr_read()`
and
`fn_fav_csr_write()`
which constitute the basic read/write actions.

Then see how these functions are used by the interface methods.

Exercise break

Please see Appendix E, Exercise Ex-08-B-CSRs.

End