Learn RISC-V CPU Implementation and BSV
(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L10: RISC-V: The Drum CPU

bluespec

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

BSV topics

Packages	Ch 3	imports, exports
Combinational circuits	Ch 4	Bit-vectors, slicing, operators, integers, identifiers, booleans, functions
		simple linear FSMs
		conditionals (muxes)
Structs and Tuples	Ch 5	
Modules and Interfaces	Ch 7	Registers, Register Files, FIFOs
Sequential FSMs	Ch 9	sequences, conditionals, while-loops
Verification	Ch 11	sequences, conditionals, while-loops
BSV Rules	Ch 13	Rule conditions, atomicity, mapping rules to clocks
Concurrent Registers	Ch 17	CRegs, Pipeline and Bypass FIFOs
Suggested further study	Ch 19	Higher-order functions, Polymorphism, typeclasses, Tagged unions, pattern matching, Multiple clock domains, Bluecheck

RISC-V examples and topics

Ch 2	RISC-V ISA	
	Recognize a legal opcode	
	Simple testbench	
	Classify an instruction	
	Memory requests/responses	
Ch 6	RISC-V core combinational functions	fn_Fetch, fn_Decode, fn_Dispatch, fn_Exec_Int, fn_Exec_Control
Ch 8	GPRs and CSRs	
Ch 10	Drum CPU	Trap-handling
Ch 12	CPU verification	
Ch 14	Drum CPU with Rules instead of StmtFSM	
Ch 15-16	Fife CPU	PC prediction, Register R/W hazards, Concurrent execution pipelines, Speculative memory access, In-order retire and trap-handling
Ch 18	Fife and Drum Optimization	Eliminate pipeline bubbles, Fuse rules (reduce cycles), Reduce state, improve PC prediction, Bypassing, Caches
Ch 20	Suggested further study	RV64I, M, A, F,D, C extensions, M,S,U privilege levels, Interrupts, Virtual memory, PMPs and protection, Memory systems, Linux-capability

Table of Contents

Drum CPU: Module Interface (same for Drum and Fife)

Drum CPU: Module Interface

```
───────────── src_Common/CPU_IFC.bsv: line 27 ... ─────────────
interface CPU_IFC;
   method Action init (Initial_Params initial_params);

   // IMem
   interface FIFOF_O #(Mem_Req) fo_IMem_req;
   interface FIFOF_I #(Mem_Rsp) fi_IMem_rsp;
   ...
   // DMem, non-speculative
   interface FIFOF_O #(Mem_Req) fo_DMem_req;
   interface FIFOF_I #(Mem_Rsp) fi_DMem_rsp;

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

The four FIFOs are for IMem and DMem requests (to memory) and IMem and DMem responses (from memory).

The init method is used to set the starting PC value (and other initial values).

The set_TIME method is used to update the time CSR.

Drum CPU: Overall module structure and state

Drum CPU: Overall module structure

```
(* synthesize *)
module mkCPU (CPU_IFC);
   // ================================================================
   // STATE
   ...

   // ****************************************************************
   // BEHAVIOR
   ... help functions ...
   ... major actions  ...

   // ================================================================
   // BEHAVIOR: FSM or Rules

`include "Drum_FSM.bsv"

   // ****************************************************************
   // INTERFACE
   ...
endmodule
```

Details in slides that follow

Drum CPU: Module state (1/3)

```
                    ───── From src_Drum/CPU.bsv ─────
  // Don't run until the PC (and other things) are initialized
  Reg #(Bool) rg_running <- mkReg (False);

  // For debugging in simulation only
  Reg #(File) rg_flog <- mkReg (InvalidFile);

  Reg #(Bit #(64)) rg_inum <- mkReg (0);    // For debugging only

  // The Program Counter
  Reg #(Bit #(XLEN)) rg_pc   <- mkReg (0);

  // General-Purpose Registers (GPRs)
  GPRs_IFC #(XLEN)  gprs <- mkGPRs_synth;

  // Control-and-Status Registers (CSRs)
  CSRs_IFC csrs <- mkCSRs;
```

... more ...

Drum CPU: Module state (2/3)

```
                        From src_Drum/CPU.sv
  // Inter-step registers
  Reg #(Fetch_to_Decode)     rg_Fetch_to_Decode     <- mkRegU;
  Reg #(Decode_to_RR)        rg_Decode_to_RR        <- mkRegU;
  Reg #(Result_Dispatch)     rg_Dispatch            <- mkRegU;
  Reg #(EX_Control_to_Retire) rg_EX_Control_to_Retire <- mkRegU;
  Reg #(EX_to_Retire)        rg_EX_to_Retire        <- mkRegU;

  // Regs to set up exception handling
  Reg #(Bool)       rg_exception <- mkReg (False);
  Reg #(Bit #(XLEN)) rg_epc       <- mkRegU;
  Reg #(Bit #(4))    rg_cause     <- mkRegU;
  Reg #(Bit #(XLEN)) rg_tval      <- mkRegU;
```

... more ...

- Inter-step registers hold intermediate values across clock cycles.
- Exception registers hold values across clocks until we perform the exception (in the final step).

Drum CPU: Module state (3/3)

```
                    ───────── From src_Drum/CPU.bsv ─────────
  // Paths to and from memory
  FIFOF #(Mem_Req) f_IMem_req  <- mkFIFOF;
  FIFOF #(Mem_Rsp) f_IMem_rsp  <- mkFIFOF;

  FIFOF #(Mem_Req) f_DMem_req  <- mkFIFOF;
  FIFOF #(Mem_Rsp) f_DMem_rsp  <- mkFIFOF;
```

Drum CPU: Some help-functions

Drum CPU: Some help-functions (1/3)

(Help-functions just encapsulate a few actions that are repeated in several places.)

This function writes a result of an instruction (a value) into the `rd` register:

```
_____ src_Drum/CPU.bsv: line 124 ... _____
function Action fa_update_rd (RR_to_Retire x1,
                             Bit #(XLEN) rd_val);
    action
       if (x1.has_rd) begin
          let rd = instr_rd (x1.instr);
          gprs.write_rd (rd, rd_val);
          ...

       end
    endaction
endfunction
```

Drum CPU: Some help-functions (2/3)

(Help-functions just encapsulate a few actions that are repeated in several places.)

This function increments the PC and the instruction-number, as we retire each instruction:

```
───────────────── src_Drum/CPU.bsv: line 138 ... ─────────────────
   function Action fa_redirect_Fetch (Bit #(XLEN) next_pc);
       action
          rg_pc    <= next_pc;
          rg_inum <= rg_inum + 1;
       endaction
   endfunction
```

Drum CPU: Some help-functions (3/3)

(Help-functions just encapsulate a few actions that are repeated in several places.)

This function records information for later trap-handling:

```
────────────────── src_Drum/CPU.bsv: line 145 ... ──────────────────
  function Action fa_setup_exception (Bit #(XLEN) epc,
                                      Bit #(4)    cause,
                                      Bit #(XLEN) tval);
     action
        rg_exception <= True;
        rg_epc       <= epc;
        rg_cause     <= cause;
        rg_tval      <= tval;
     endaction
  endfunction
```

Drum CPU: top-level FSM

Drum CPU: top-level FSM (1/2)

```
                    ── src_Drum/Drum_FSM.bsv: line 41 ... ──
  mkAutoFSM (seq
                await (rg_running);
                while (True) exec_one_instr;
            endseq);
```

The very top-level is very simple and self-evident.

The await statement waits until the init method has set the initial PC, so that we start fetching from the correct address.

Drum CPU: top-level FSM (2/2)

```
_____ src_Drum/Drum_FSM.bsv: line 10 ... _____
Stmt exec_one_instr =
seq
   a_Fetch;
   a_Decode;
   a_Register_Read_and_Dispatch;

   // Execute and Retire
   if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_DIRECT)
      a_Retire_direct;
   else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_CONTROL)
      seq    // BRANCH, JAL, JALR
         a_EX_Control;
         a_Retire_Control;
      endseq
   else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_INT)
      seq    // LUI, AUIPC, IALU
         a_EX_Int;
         a_Retire_Int;
      endseq
   else if (rg_Dispatch.to_Retire.exec_tag == EXEC_TAG_DMEM)
      seq
         a_EX_DMem;
         a_Retire_DMem;
      endseq
   else    // IMPOSSIBLE
      noAction;

   if (rg_exception)
      a_exception;
endseq;
```

This is practically a direct coding of the instruction-execution steps in the diagram on Slide 4.

Drum CPU: FSM Fetch step

```
_____ src_Drum/CPU.bsv: line 160 ... _____
Action a_Fetch =
action
   ...
   let y <- fn_Fetch (rg_pc,
                      ...
   rg_Fetch_to_Decode <= y.to_D;
   f_IMem_req.enq (y.mem_req);
   ...
endaction;
```

Just invokes fn_Fetch
and sends the two re-
sults into a register for
Fetch and into a FIFO
for IMem memory.

Drum CPU: FSM Decode step

```
                    src_Drum/CPU.bsv: line 175 ...
   Action a_Decode =
   action
      let mem_rsp <- pop_o (to_FIFOF_O (f_IMem_rsp));
      let y        <- fn_Decode (rg_Fetch_to_Decode, mem_rsp, rg_flog);
      rg_Decode_to_RR <= y;
      ...
   endaction;
```

Just invokes fn_Decode
on the inputs from Fetch
and memory, and sends
the result into a reg-
ister for Register-Read-
and-Dispatch.

Drum CPU: FSM Register-Read and Dispatch step

```
──────────── src_Drum/CPU.bsv: line 185 ... ────────────
Action a_Register_Read_and_Dispatch =
action
   // Read GPRs
   // Ok that read_rs1 and read_rs2 may return junk values
   //         since not all instrs have rs1/rs2.
   let x      = rg_Decode_to_RR;
   let rs1_val = gprs.read_rs1 (instr_rs1 (x.instr));
   let rs2_val = gprs.read_rs2 (instr_rs2 (x.instr));

   Result_Dispatch y <- fn_Dispatch (x, rs1_val, rs2_val, rg_flog);
   rg_Dispatch        <= y;
   ...
endaction;
```

Using input from Decode, reads two registers rs1 and rs2; then invokes fn_Dispatch, and and sends the result into a register for the Execute stage.

Drum CPU: FSM Execute and Retire step

Drum CPU: FSM Execute and Retire step flows
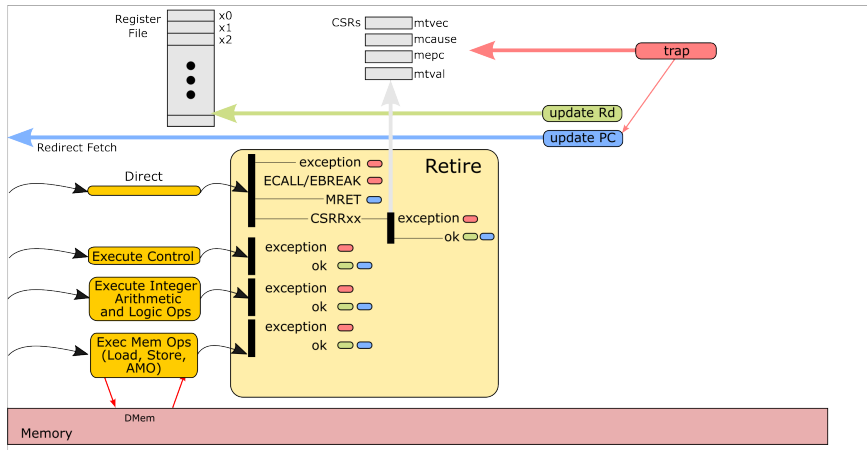
4 possible flows.

- Direct
- Control
- Int
- DMem

Each ultimately resulting in up to 3 possible actions.

Drum CPU: Updating the `minstret` CSR (counting instructions retired)

The `minstret` CSR is incremented for each instruction retired.
Except:

- An instruction that raises an exception.
- A CSRRxx instruction that writes a value into `minstret`.

We increment using:

    csrs.ma_incr_instret;

The `minstret` and `mcycle` CSRs are useful in performance measurement (instructions/cycle).

Drum CPU: Execute and Retire Direct flow (1/4)

```
                        ──── From src_Drum/CPU.bsv ────
  Action a_Retire_direct =
  action
     let x_direct = rg_Dispatch.to_Retire;
     if (x_direct.exception) begin
        fa_setup_exception (x_direct.pc,        // epc
                            x_direct.cause,     // cause
                            x_direct.tval);     // tval
        log_Retire_Direct_exc (rg_flog, x_direct);
     end
     // ---------------
     ...
```

Drum CPU: Execute and Retire Direct flow (2/4)

```
                    ─── From src_Drum/CPU.bsv ───
    ...
    // ----------------
    else if (is_legal_CSRRxx (x_direct.instr)) begin
       match { .exc, .rd_val } <- csrs.mav_csrrxx (x_direct.instr,
                                                   x_direct.rs1_val);
       if (exc)
          fa_setup_exception (x_direct.pc,                 // epc
                              cause_ILLEGAL_INSTRUCTION,   // cause
                              x_direct.instr);             // tval
       else begin
          fa_update_rd (x_direct, rd_val);
          fa_redirect_Fetch (x_direct.fallthru_pc);
       end
       log_Retire_CSRRxx (rg_flog, exc, x_direct);
    end
    // ----------------
    ...
```

Drum CPU: Execute and Retire Direct flow (3/4)

```
                      From src_Drum/CPU.bsv
      ...
      // ----------------
      else if (is_legal_MRET (x_direct.instr)) begin
         fa_redirect_Fetch (csrs.read_epc);
         csrs.ma_incr_instret;
         log_Retire_MRET (rg_flog, x_direct);
      end
      // ----------------
      ...
```

```
                          ── From src_Drum/CPU.bsv ──
     ...
     // ----------------
     else if (is_legal_ECALL (x_direct.instr)
             || is_legal_EBREAK (x_direct.instr))
       begin
         let cause = ((x_direct.instr [20] == 0)
                        ? cause_ECALL_FROM_M
                        : cause_BREAKPOINT);
         fa_setup_exception (x_direct.pc,    // epc
                             cause,
                             0);             // tval
         csrs.ma_incr_instret;
         log_Retire_ECALL_EBREAK (rg_flog, x_direct);
       end
     else begin
       wr_log2 (rg_flog, $format ("CPU.EX.Direct: IMPOSSIBLE"));
       $finish (1);
     end
  endaction;
```

Drum CPU: Execute and Retire Control flow

──────────── From src_Drum/CPU.bsv ────────────

```
Action a_EX_Control =
action
   let x = rg_Dispatch.to_EX_Control;
   let y <- fn_EX_Control (x, rg_flog);
   rg_EX_Control_to_Retire <= y;
   ...
endaction;
```

```
Action a_Retire_Control =
action
   let x_direct  = rg_Dispatch.to_Retire;
   let x_control = rg_EX_Control_to_Retire;
   if (x_control.exception)
      fa_setup_exception (x_direct.pc,
                          x_control.cause,
                          x_control.tval);
   else begin
      fa_update_rd (x_direct, x_control.data);
      fa_redirect_Fetch (x_control.next_pc);
      csrs.ma_incr_instret;
   end
   ...
endaction;
```

Drum CPU: Execute and Retire Int flow

```
                        ── From src_Drum/CPU.bsv ──
   Action a_EX_Int =
   action
      let x = rg_Dispatch.to_EX;
      let y <- fn_EX_Int (x, rg_flog);
      rg_EX_to_Retire <= y;
      ...
   endaction;
```

```
   Action a_Retire_Int =
   action
      if (rg_EX_to_Retire.exception)
         fa_setup_exception (rg_Dispatch.to_Retire.pc,
                             rg_EX_to_Retire.cause,
                             rg_EX_to_Retire.tval);
      else begin
         fa_update_rd (rg_Dispatch.to_Retire, rg_EX_to_Retire.data);
         fa_redirect_Fetch (rg_Dispatch.to_Retire.fallthru_pc);
         csrs.ma_incr_instret;
      end
      ...
   endaction;
```

```
                        ── From src_Drum/CPU.bsv ──
  Action a_EX_DMem =
  action
     Mem_Req y = rg_Dispatch.to_EX_DMem;
     f_DMem_req.enq (y);
     ...
  endaction;
```

Drum CPU: Execute and Retire DMem flow (2/2)

```
                         ───── From src_Drum/CPU.bsv ─────
Action a_Retire_DMem =
action
   let x_direct = rg_Dispatch.to_Retire;
   let mem_rsp <- pop_o (to_FIFOF_O (f_DMem_rsp));
   Bool exception = ((mem_rsp.rsp_type == MEM_RSP_ERR)
                     || (mem_rsp.rsp_type == MEM_RSP_MISALIGNED));
   if (exception) begin
      Bit #(4) cause = ((mem_rsp.rsp_type == MEM_RSP_MISALIGNED)
                        ? (is_LOAD (x_direct.instr)
                           ? cause_LOAD_ADDRESS_MISALIGNED
                           : cause_STORE_AMO_ADDRESS_MISALIGNED)
                        : (is_LOAD (x_direct.instr)
                           ? cause_LOAD_ACCESS_FAULT
                           : cause_STORE_AMO_ACCESS_FAULT));
      fa_setup_exception (x_direct.pc,                // epc
                          cause,
                          truncate (mem_rsp.addr));   // tval
   end
   else begin
      fa_update_rd (x_direct, truncate (mem_rsp.data));
      fa_redirect_Fetch (x_direct.fallthru_pc);
      csrs.ma_incr_instret;
   end
   ...
endaction;
```

Drum CPU: action for exceptions

```
                          From src_Drum/CPU.bsv
Action a_exception =
action
   Bool is_interrupt = False;
   Bit #(XLEN) tvec_pc <- csrs.mav_exception (rg_epc,
                                              is_interrupt,
                                              rg_cause,
                                              rg_tval);

   rg_exception <= False;
   fa_redirect_Fetch (tvec_pc);
   ...
endaction;
```

Drum CPU: final comments

Drum CPU code, using StmtFSM, looks just like a C/C++ program with somewhat different syntax.

So: can't we just code it in C/C++ and compile that to hardware?

- Difficult to compile *general-purpose* C into hardware.
  C has many constructs that have no obvious mapping into hardware, such as pointers, the address-of operator "&", address arithmetic, malloc, and so on.
  But can't we define a restricted subset of C suitable for hardware design?
  Compilation is still very difficult: no distinction between variables used to hold state (registers) *vs.* variables used to hold temporary intermediate values (wires); no distinction between pure and side-effect constructs; ...

- No concurrency. It may be possible to compile a subset of C to hardware, similar to Drum. But this is the easy case (completely sequential). Very hard to generalize to pipelines and more concurrent implementations (Fife, and beyond).

End