

Learn RISC-V CPU Implementation and **BSV**

(**BSV**: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L6: RISC-V: Core functions for ISA execution



Reminders

Please git clone: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design
(git pull for latest version). Repository structure:

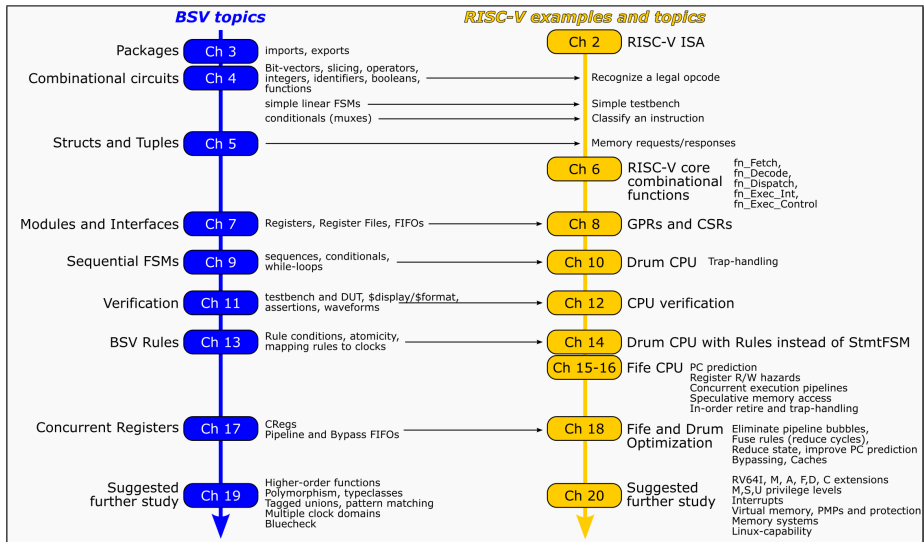
```
./Book_BLang_RISCV.pdf
  Slides/
    Slides_01_Intro.pdf
    Slides_02_ISA.pdf
    ...
  Exercises/
    Ex-03-A-Hello-World/
    Ex-03-B-Top-and-DUT/
    ...
  Code/
    src_Top/
    src_Drum/
    src_Fife/
    src_Common/
    ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- Slides and Exercise are numbered in sync with book Chapter numbers.
- For Exercises, please see Appendix E of the book. Some (not all) exercises have associated code in the Exercises/ directory.

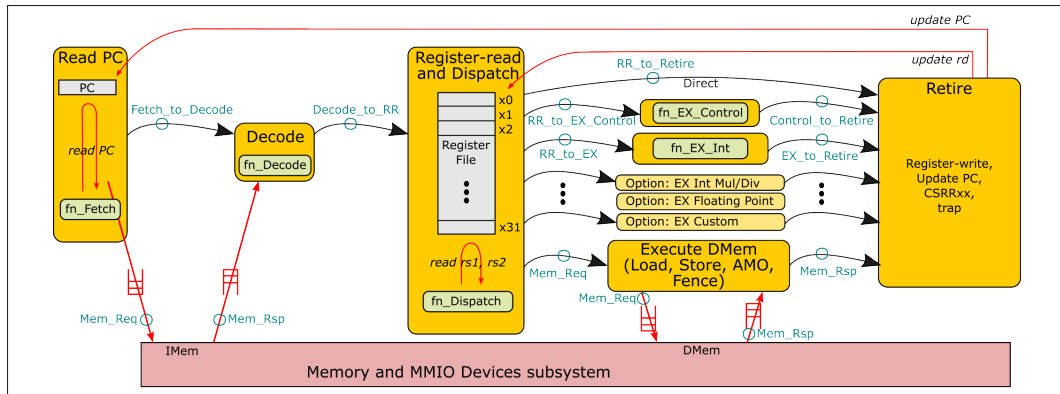
To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- 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



The green annotations indicate the type of information flowing on each arrow.

Each of these is a “struct” type (also known as a “record”): a heterogeneous grouping of *fields* of various types.

Table of Contents

- 1 Fetch
- 2 Decode
- 3 Dispatch
- 4 Execute Control
- 5 Execute Int

fn_Fetch

Excerpts shown in next few slides.

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

Inputs and outputs of fn_Fetch

Direct output from fn_Fetch to Decode:

```
src_Common/Inter_Stage.bsv: line 27 ...  
typedef struct {  
    Bit #(XLEN)  pc;  
    ...  
    Bit #(64)    inum;           // for debugging only  
} Fetch_to_Decode  
deriving (Bits, FShow);
```

Overall output of fn_Fetch

```
src_Common/Fn.Fetch.bsv: line 26 ...  
typedef struct {  
    Fetch_to_Decode  to_D;  
    Mem_Req          mem_req;  
} Result_F  
deriving (Bits, FShow);
```

fn_Fetch

src_Common/Fn_Fetch.bsv: line 34 ...

```
function ActionValue #(Result_F)
  fn_Fetch (Bit #(XLEN) pc,
            ...
            Bit #(64) inum,
            ...

  actionvalue
    Result_F y = ?;
    // Info to next stage
    y.to_D = Fetch_to_Decode {pc:          pc,
                              ...
                              // Request to IMem
                              y.mem_req = Mem_Req {req_type: funct5_LOAD,
                                                    size:      MEM_4B,
                                                    addr:      zeroExtend (pc),
                                                    data :      ?,
                                                    // Debugging
                                                    inum:      inum,
                                                    ...

    return y;
  endactionvalue
endfunction
```


fn_Fetch is actually a pure function

fn_Fetch is actually a pure (side effect-free) function and could have been written without ActionValue#():

```
function ActionValue #(Result_F)
    fn_Fetch (Bit #(XLEN)  pc,
              ...
              Bit #(64)    inum);
    actionvalue
        Result_F y = ?;
    ...
    return y;
endactionvalue
endfunction
```

⇒

```
function Result_F
    fn_Fetch (Bit #(XLEN)  pc,
              ...
              Bit #(64)    inum);
    Result_F y = ?;
    ...
    return y;
endfunction
```

We write it this way only to make it easy to add `$display()` statements for debugging in case we need them (when `ActionValue#()` would be necessary).



Exercise break

Please see Appendix E, Section Ex-06-A-Fetch.

fn_Decode

Excerpts shown in next few slides.

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

Output of fn_Decode

```
_____ src_Common/Inter_Stage.bsv: line 48 ... _____
typedef struct {Bit #(XLEN)  pc;

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

                // If not exception
                Bit #(XLEN)   fallthru_pc;
                Bit #(32)     instr;
                OpClass       opclass;
                Bool          has_rs1;
                Bool          has_rs2;
                Bool          has_rd;
                Bool          writes_mem; // All mem ops other than LOAD
                Bit #(XLEN)   imm;       // Canonical (bit-swizzled)

                ...
} Decode_to_RR
deriving (Bits, FShow);
```

Fields of struct Decode_to_RR

- The current PC: will be needed by BRANCH, JAL, AUIPC to compute a target address relative to the current PC.
- Was there an exception (Fetch memory error, or instruction is not legal)? If so, what was the cause?
- If no exception, what is the fall-through PC (needed for next-PC update for most instructions, and for saved-PC (“return address”) for JAL/JALR).
- What is the instruction? What is its broad classification: Control (Branch or Jump)? Integer Arithmetic or Logic? Memory Access? This will help in choosing the execute stage pipeline.
- Does it have zero, one or two input registers (“rs1” and “rs2”)? If so, which ones? This will help the Register-Read stage in reading registers and, for Fife, for managing “hazards”.
- Does it have zero or one output registers (“rd”)? If so, which one? This will help the final Register Write stage in writing back a value to a register.
- Does it write memory? In Fife, where we do speculative writes to memory, this will help the Retire stage commit (finalize) those writes.
- What is the “immediate” value, if any (after untangling all different ways in which bits are permuted in different formats of instructions).

Fields of struct Decode_to_RR

Some Decode_to_RR fields, such as `has_rs1`, are used in later stages, and could be computed there from `instr`; so why compute them here in Decode?

- If something is needed in more than one stage, computing it once and carrying the value forward can save some hardware cost (gates). It also incurs a hardware cost: we have to allocate state elements for the carried value in each inter-stage buffer/FIFO.
- Wherever something is computed, it adds to combinational delay for that stage (lowering achievable clock speed).

The decision between compute-and-carry vs. compute later is a balancing act between these kinds of considerations.

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



Exercise break

Please see Appendix E, Section Ex-06-B-Decode.

fn_Dispatch

Excerpts shown in next few slides.

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

Partial output of fn_Dispatch: direct to Retire

```
src_Common/Inter_Stage.bsv: line 81 ...
typedef struct {Exec_Tag      exec_tag;    // ‘‘flow’’ for this instr

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

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

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

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

Partial output of fn_Dispatch: to Execute-Control

```
src_Common/Inter_Stage.bsv: line 109 ...  
typedef struct {Bit #(XLEN)  pc;  
                  Bit #(XLEN)  fallthru_pc;  
                  Bit #(32)    instr;  
                  Bit #(XLEN)  rs1_val;  
                  Bit #(XLEN)  rs2_val;  
                  Bit #(XLEN)  imm;  
                  Bit #(64)    inum;    // for debugging only  
} RR_to_EX_Control  
deriving (Bits, FShow);
```

Partial output of fn_Dispatch: to Execute-Integer ALU

```
src_Common/Inter_Stage.bsv: line 140 ...  
typedef struct {Bit #(32)    instr;  
                  Bit #(XLEN) rs1_val;  
                  Bit #(XLEN) rs2_val;  
                  Bit #(XLEN) imm;  
                  ...  
} RR_to_EX  
deriving (Bits, FShow);
```

Partial output of fn_Dispatch: to Execute-Integer ALU

```
src_Common/Inter_Stage.bsv: line 140 ...  
typedef struct {Bit #(32)    instr;  
                  Bit #(XLEN) rs1_val;  
                  Bit #(XLEN) rs2_val;  
                  Bit #(XLEN) imm;  
                  ...  
} RR_to_EX  
deriving (Bits, FShow);
```

Full output of fn_Dispatch

src.Common/Fn_Dispatch.bsv: line 30 ...

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

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



Exercise break

Please see Appendix E, Section Ex-06-C-Dispatch.

fn_Ex_Control

Excerpts shown in next few slides.

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

Output of fn_Ex_Control

```
src_Common/Inter_Stage.bsv: line 120 ...  
typedef struct {Bool      exception; // Misaligned BRANCH/JAL/JALR target  
                  Bit #(4)  cause;  
                  Bit #(XLEN) tval;  
  
                  Bit #(XLEN) next_pc;  
                  Bit #(XLEN) data;          // Return-PC for JAL/JALR  
                  ...  
} EX_Control_to_Retire  
deriving (Bits, FShow);
```

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



Exercise break

Please see Appendix E, Section Ex-06-D-Ex-Control.

fn_Ex_Int

Excerpts shown in next few slides.

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

Output of fn_Ex_Int

```
src_Common/Inter_Stage.bsv: line 153 ...  
typedef struct {Bool      exception;  
                  Bit #(4)  cause;  
                  Bit #(XLEN) tval;  
  
                  Bit #(XLEN) data;  
                  ...  
} EX_to_Retire  
deriving (Bits, FShow);
```

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



Exercise break

Please see Appendix E, Section Ex-06-E-Ex-Int.

End