

Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L5: **BSV** Structs; Memory requests and responses



Reminders

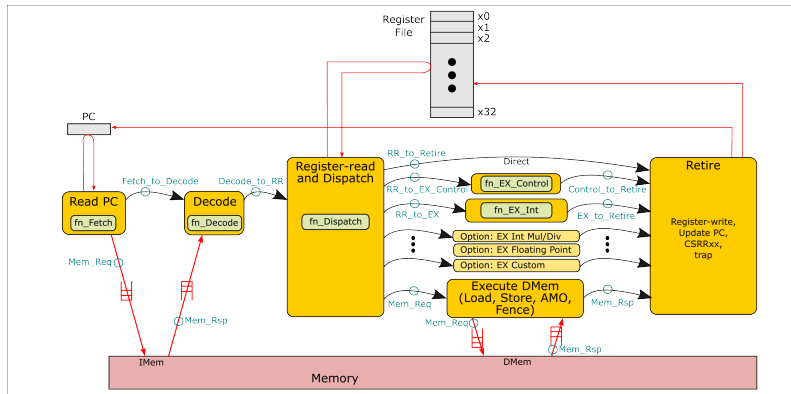
Please git clone or git pull: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design

```
./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/>)

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.

Output from fn_Decode to the next stage

```
_____ 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.

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?

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).

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.

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”.

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.

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.

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.

BSV: about “deriving (Bits, FShow)”

- Because we said “deriving (Bits)”, *bsc* will automatically pick a hardware representation of this struct as a bit-vector, by simply concatenating the bit-representations of the fields.
- If we wanted a different representation, we omit “deriving (Bits)”, and there is a way (“Typeclass Instances”) to specify exactly what we want.

BSV: about “deriving (Bits, FShow)”

- Because we said “deriving (Bits)”, *bsc* will automatically pick a hardware representation of this struct as a bit-vector, by simply concatenating the bit-representations of the fields.
- If we wanted a different representation, we omit “deriving (Bits)”, and there is a way (“Typeclass Instances”) to specify exactly what we want.
- Because we said “deriving (FShow)”, *bsc* will automatically define an “fshow()” function for this struct, that will print each field separately. Otherwise `$display()` will simply print the flat bit-vector representation of the struct (which will likely be hard to read).
- If we wanted way to print the structa, we omit “deriving (FShow)”, and there is a way (“Typeclass Instances”) to define `fshow()` to print what we want.

BSV: struct expressions to construct struct values

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

The right-hand side is a “struct expression” whose value is a struct value.

If a field is left undefined, *bsc* will warn while compiling.

BSV: struct expressions to construct struct values

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

The right-hand side is a “struct expression” whose value is a struct value.

If a field is left undefined, *bsc* will warn while compiling.

Some shorthands: “let” and don’t-care values:

```
let x = Decode_to_RR {pc:      ... ,  
                        exception: False,  
                        cause:    ?,  
                        fallthru_pc: ... ,  
                        instr:    ... ,  
                        ...};
```

BSV: Accessing and updating struct fields

These notations are standard in many programming languages.

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

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

BSV: Tuples: pre-defined structs with special notation

Example (from `mkCSRs` module in file `src_Common/CSRs.bsv`).

Constructing a 2-tuple value:

```
function ActionValue #(Tuple2 #(Bool, Bit #(XLEN)))  
    fav_csr_read (Bit #(12) csr_addr);  
    ...  
    return tuple2 (exception, y);  
endfunction
```

BSV: Tuples: pre-defined structs with special notation

Example (from `mkCSRs` module in file `src/Common/CSRs.bsv`).

Constructing a 2-tuple value:

```
function ActionValue #(Tuple2 #(Bool, Bit #(XLEN)))  
  fav_csr_read (Bit #(12) csr_addr);  
  ...  
  return tuple2 (exception, y);  
endfunction
```

Accessing struct components using predefined functions `tpl_j`:

```
let xy <- fav_csr_read (...);  
let exc = tpl_1 (xy);    // exc has type: Bool  
let v   = tpl_2 (xy);    // v   has type: Bit #(XLEN)
```

BSV: Tuples: pre-defined structs with special notation

Example (from `mkCSRs` module in file `src/Common/CSRs.bsv`).

Constructing a 2-tuple value:

```
function ActionValue #(Tuple2 #(Bool, Bit #(XLEN)))  
  fav_csr_read (Bit #(12) csr_addr);  
  ...  
  return tuple2 (exception, y);  
endfunction
```

Accessing struct components using predefined functions `tpl_j`:

```
let xy <- fav_csr_read (...);  
let exc = tpl_1 (xy);    // exc has type: Bool  
let v   = tpl_2 (xy);    // v   has type: Bit #(XLEN)
```

Accessing struct components using pattern-matching:

```
match { .exc, .v } <- fav_csr_read (csr_addr);
```

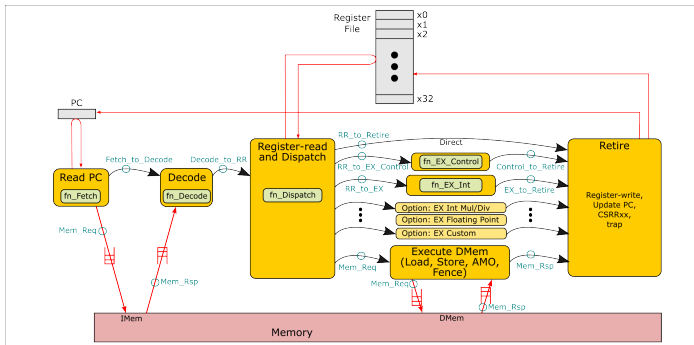
“Harvard” Architecture: Separating Instruction and Data Memory

Since the very days of computers, most computers have separate channels to memory:

- “IMem”: for Fetch to read from instruction memory
- “DMem”: for LOAD/STORE instructions to read/write data memory

Typically, IMem and DMem can be accessed concurrently.

To facilitate this concurrency, programs typically do not modify their own instructions with LOAD/STORE instructions.



This organization is sometimes loosely called a “Harvard Architecture”.
See: https://en.wikipedia.org/wiki/Harvard_architecture.

Memory requests

```
src_Common/Mem_Req_Rsp.bsv: line 44 ...  
  
typedef struct {Mem_Req_Type req_type;  
                Mem_Req_Size size;  
                Bit #(64)    addr;  
                Bit #(64)    data;    // CPU => mem data  
                ...  
} Mem_Req  
deriving (FShow, Bits);
```

We will discuss Mem_Req_Type in Mem_Req_Size the following slides.

The data field is only relevant when communicating data from the CPU to memory (STORE and AMO instructions). When communicating 1, 2 and 4 bytes, these are in the least-significant bytes of the data field.

Memory requests

```
src_Common/Mem_Req_Rsp.bsv: line 44 ...  
  
typedef struct {Mem_Req_Type req_type;  
                Mem_Req_Size size;  
                Bit #(64)    addr;  
                Bit #(64)    data;    // CPU => mem data  
                ...  
} Mem_Req  
deriving (FShow, Bits);
```

We will discuss Mem_Req_Type in Mem_Req_Size the following slides.

The data field is only relevant when communicating data from the CPU to memory (STORE and AMO instructions). When communicating 1, 2 and 4 bytes, these are in the least-significant bytes of the data field.

Note, we do not say “deriving (Eq)” because we have no occasion to compare two entire Memory Requests for equality/inequality.

Memory requests: Mem_Req_Type

We could define Mem_Req_Type as an enum type (MEM_REQ_LOAD, MEM_REQ_STORE ...). However, looking ahead to possibly supporting the “A” extension in the future (Atomic Memory Ops (AMOs), see Unprivileged ISA Spec p.132), we observe that the ISA defines a 5-bit code for each AMO op. So, we define:

```
src_Common/Mem_Req_Rsp.bsv: line 16 ...  
typedef Bit #(5) Mem_Req_Type;
```

For the AMOs, we will use the 5-bit codes as defined in the ISA (simplifies Decode!).
For LOAD/STORE/FENCE, we use three codes that are not used by AMO ops.

```
src_Common/Instr_Bits.bsv: line 226 ...  
Bit #(5) funct5_LOAD      = 5'b_11110;  
Bit #(5) funct5_STORE     = 5'b_11111;  
Bit #(5) funct5_FENCE     = 5'b_11101;
```

Memory requests: Mem_Req_Size

```
src_Common/Mem_Req_Rsp.bsv: line 41 ...  
typedef enum {MEM_1B, MEM_2B, MEM_4B, MEM_8B} Mem_Req_Size  
deriving (Eq, FShow, Bits);
```

Memory requests: Mem_Req_Size

```
src_Common/Mem_Req_Rsp.bsv: line 41 ...  
typedef enum {MEM_1B, MEM_2B, MEM_4B, MEM_8B} Mem_Req_Size  
deriving (Eq, FShow, Bits);
```

Why do we have a code for 8 bytes, since RV32I can only LOAD/STORE bytes (1 byte), halfwords (2 bytes) and words (4 bytes)?

Memory requests: Mem_Req_Size

```
src_Common/Mem_Req_Rsp.bsv: line 41 ...  
typedef enum {MEM_1B, MEM_2B, MEM_4B, MEM_8B} Mem_Req_Size  
deriving (Eq, FShow, Bits);
```

Why do we have a code for 8 bytes, since RV32I can only LOAD/STORE bytes (1 byte), halfwords (2 bytes) and words (4 bytes)?

This is with an eye towards future extension of our implementation:

- If we support the D ISA extension (double-precision floating point), we'll need to be able to load/store doublewords (8 bytes)
- If we support RV64I, we'll need to be able to load/store doublewords (8 bytes).

Memory requests: Mem_Req_Size

```
src_Common/Mem_Req_Rsp.bsv: line 41 ...  
typedef enum {MEM_1B, MEM_2B, MEM_4B, MEM_8B} Mem_Req_Size  
deriving (Eq, FShow, Bits);
```

Why do we have a code for 8 bytes, since RV32I can only LOAD/STORE bytes (1 byte), halfwords (2 bytes) and words (4 bytes)?

This is with an eye towards future extension of our implementation:

- If we support the D ISA extension (double-precision floating point), we'll need to be able to load/store doublewords (8 bytes)
- If we support RV64I, we'll need to be able to load/store doublewords (8 bytes).
- Even though RV32I and RV64I instructions are at most 32-bits wide, the Fetch stage may choose to fetch 64 bits at a time, effectively “pre-fetching” a next instruction and reducing the number of memory accesses.



Exercise break

Please see directory: `Exercises/Ex_05_A_Structs/`
and its README.

Memory responses

Memory responses may report an exception (misaligned access, non-existent memory, ...).

```
src_Common/Mem_Req_Rsp.bsv: line 58 ...  
  
typedef enum {MEM_RSP_OK,  
              MEM_RSP_MISALIGNED,  
              MEM_RSP_ERR,  
              ...  
} Mem_Rsp_Type  
deriving (Eq, FShow, Bits);
```

```
src_Common/Mem_Req_Rsp.bsv: line 67 ...  
  
typedef struct {Mem_Rsp_Type  rsp_type;  
                Bit #(64)     data;      // mem => CPU data  
                ...  
} Mem_Rsp  
deriving (FShow, Bits);
```

The data field is only relevant when communicating data from memory to the CPU (LOAD and LR instructions). When communicating 1, 2 and 4 bytes, these are in the least-significant bytes of the data field.



Exercise break

Please see directory: `Exercises/Ex_05_B_Mem_Req_Rsp/`
and its README.

End