

# 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

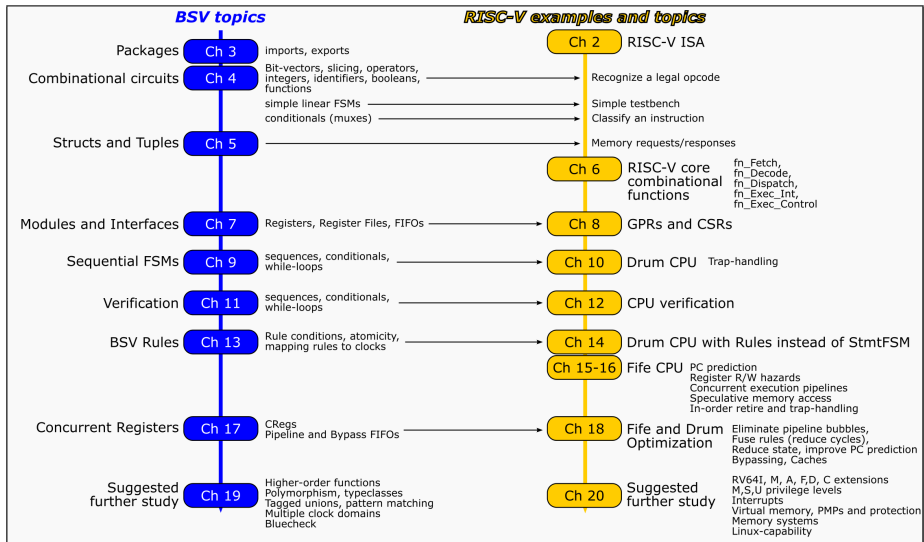
Please git clone or git pull: [https://github.com/rsnikhil/Learn\\_Bluespec\\_and\\_RISCV\\_Design](https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design)

```
./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/
    ...
```

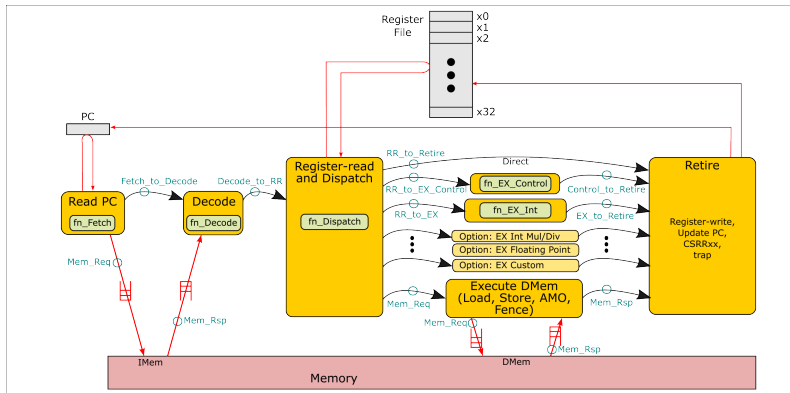
To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- *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



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 grouping of *fields* of heterogeneous types.

# 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 (Bits, FShow);
```

We will discuss Mem\_Req\_Type in Mem\_Req\_Size some following slides (they are just small bit-width scalars).

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.

# BSV: struct expressions to construct struct values

```
Mem_Req x = Mem_Req {req_type: ... value of field ... ,  
                    size:    ... value of field ... ,  
                    addr:    ... value of field ... ,  
                    data:    ... 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 = Mem_Req {req_type: MEM_REQ_LOAD,  
                size:    ... value of field ... ,  
                addr:    ... value of field ... ,  
                data:    ?,  
                ...    };
```

# BSV: Accessing and updating struct fields

These notations are standard in many programming languages.

```
x.req_type  
x.size
```

```
x.req_type = MEM_REQ_STORE;  
x.data     = ... new value ... ;
```

# BSV: Bit-representation of struct values

- 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. So, the size of the bit-vector for a struct value is the sum of the sizes of the bit-vectors for the fields.
- If we wanted a different, custom representation, we omit “`deriving (Bits)`”, and there is a way (“`Typeclass Instances`”) to specify exactly what we want.



# BSV: printing/logging struct values (for debugging)

We can print a struct value directly, *e.g.*,

```
Mem_Req mem_req;  
...  
$display ("mem_req is: ", mem_req);
```

This will just print the hexadecimal notation for the full bit-vector representing the struct. This can be difficult to read:

- Some structs are large (hundreds of bits!)
- Field boundaries may not align with hexadecimal bit boundaries (every 4 bits), and so correlating the hex digits to the fields can be tedious.

# BSV: printing/logging struct values (for debugging)

Because we said “`deriving (FShow)`”, *bsc* will automatically define an “`fshow()`” function for this struct, that will print each field separately.

```
Mem_Req mem_req;  
...  
$display ("mem_req is: ", fshow (mem_req));
```

If we wanted way to print the struct in a custom format, we omit “`deriving (FShow)`”, and there is a way (“Typeclass Instances”) to define `fshow()` to print what we want.

# 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 (Bits, FShow, Eq);
```

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). This could be in RV32 or RV64.
- 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 more bits on each memory access, effectively “pre-fetching” subsequent instruction and reducing the number of memory accesses.



## Exercise break

Please see Appendix E, Section Ex-05-A-Structs.

# 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 (Bits, FShow, Eq);
```

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

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 Appendix E, Section Ex-05-B-Mem-Req-Rsp.

# BSV: Tuples: pre-defined immutable structs with special notation

Constructing a 2-tuple value: Example:

```
_____ from src_Common/CSRs.bsv _____  
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);
```



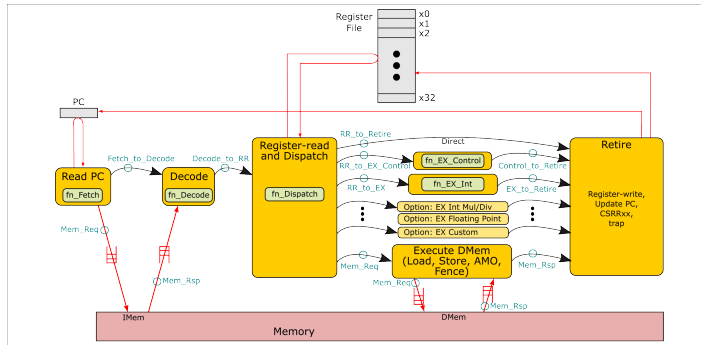
# “Harvard” Architecture: Separating Instruction and Data Memory

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](https://en.wikipedia.org/wiki/Harvard_architecture).

End