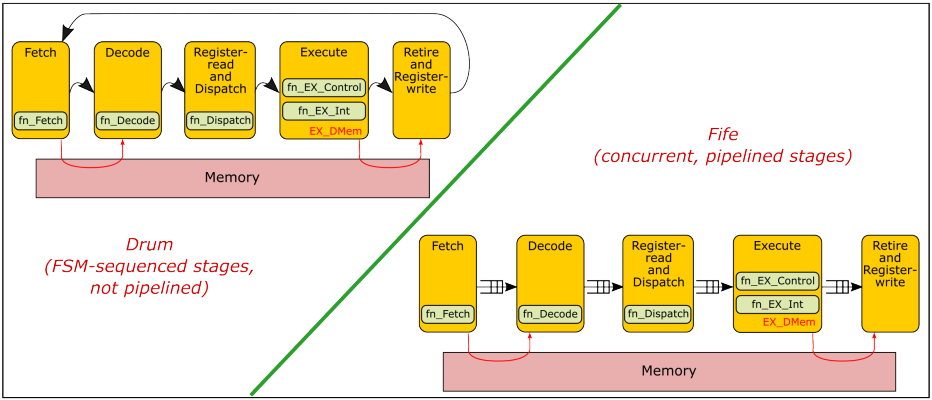
Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L4: BSV: Combinational Circuits

Two CPU implementations (microarchitectures): Drum and Fife



We start learning BSV by coding the `fn_XXX` functions. These are used in both Drum and Fife, and are all combinational circuits.

We start with `fn_Decode`).

Inputs to `fn_Decode`

The inputs to the Decode stage, as shown in the diagram are:

[1] When implementing the so-called "C" RISC-V ISA extension ("compressed instructions"), instructions can also be 16-bits, but we ignore that for now.

Inputs to `fn_Decode`

The inputs to the Decode stage, as shown in the diagram are:

- (From IMem ("instruction-memory")): A 32-bit piece of data—a RISC-V instruction—that has become available by reading it from memory at the PC address.[1]

[1] When implementing the so-called "C" RISC-V ISA extension ("compressed instructions"), instructions can also be 16-bits, but we ignore that for now.

Inputs to `fn_Decode`

The inputs to the Decode stage, as shown in the diagram are:

- (From IMem ("instruction-memory")): A 32-bit piece of data—a RISC-V instruction—that has become available by reading it from memory at the PC address.[1]
- (Direct from Fetch stage): any additional information for this instruction that did not need to go to memory and back.

[1] When implementing the so-called "C" RISC-V ISA extension ("compressed instructions"), instructions can also be 16-bits, but we ignore that for now.

Inputs to `fn_Decode`

The inputs to the Decode stage, as shown in the diagram are:

- (From IMem ("instruction-memory")): A 32-bit piece of data—a RISC-V instruction—that has become available by reading it from memory at the PC address.[1]
- (Direct from Fetch stage): any additional information for this instruction that did not need to go to memory and back.

We will use a BSV "`struct`" type (to be described soon) whenever we want to carry multiple pieces of information together.

Example: a memory request will carry a request-type (such as READ) and an address together.

[1] When implementing the so-called "C" RISC-V ISA extension ("compressed instructions"), instructions can also be 16-bits, but we ignore that for now.

Outputs from `fn_Decode`

The outputs from the Decode stage, as shown in the diagram are:

Outputs from `fn_Decode`

The outputs from the Decode stage, as shown in the diagram are:

- Was the Fetch itself successful, or did it encounter a memory error; if so, what kind of memory error?

Outputs from `fn_Decode`

The outputs from the Decode stage, as shown in the diagram are:

- Was the Fetch itself successful, or did it encounter a memory error; if so, what kind of memory error?
- Is it a legal 32-bit instruction?

Outputs from `fn_Decode`

The outputs from the Decode stage, as shown in the diagram are:

- Was the Fetch itself successful, or did it encounter a memory error; if so, what kind of memory error?
- Is it a legal 32-bit instruction?
- If legal, what is its broad classification: Control (Branch or Jump)? Integer Arithmetic or Logic? Memory Access? This will help in choosing the next stage to which we must dispatch to execute the instruction.

Outputs from `fn_Decode`

The outputs from the Decode stage, as shown in the diagram are:

- Was the Fetch itself successful, or did it encounter a memory error; if so, what kind of memory error?
- Is it a legal 32-bit instruction?
- If legal, what is its broad classification: Control (Branch or Jump)? Integer Arithmetic or Logic? Memory Access? This will help in choosing the next stage to which we must dispatch to execute the instruction.
- Does it have zero, one or two input registers ("rs1" and "rs2")? If so, which ones? This will help the next stage in reading registers.

Outputs from `fn_Decode`

The outputs from the Decode stage, as shown in the diagram are:

- Was the Fetch itself successful, or did it encounter a memory error; if so, what kind of memory error?
- Is it a legal 32-bit instruction?
- If legal, what is its broad classification: Control (Branch or Jump)? Integer Arithmetic or Logic? Memory Access? This will help in choosing the next stage to which we must dispatch to execute the instruction.
- Does it have zero, one or two input registers ("rs1" and "rs2")? If so, which ones? This will help the next stage in reading registers.
- 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.

Integer literals (constants) in BSV

Integer literals use the same notation as in Verilog and SystemVerilog:

```
3'b010            // Binary literal, 3 bits wide
7'b_110_0011      // Binary literal, 7 bits wide
5'h3              // Hex literal, 5 bits wide
32'h3             // Hex literal, 5 bits wide
32'h_efff_0f17    // Hex literal, 32 bits wide (an AUIPC instruction)
'h23              // Hex literal, context determines width
```

Integer literals (constants) in BSV

Integer literals use the same notation as in Verilog and SystemVerilog:

```
3'b010            // Binary literal, 3 bits wide
7'b_110_0011      // Binary literal, 7 bits wide
5'h3              // Hex literal, 5 bits wide
32'h3             // Hex literal, 5 bits wide
32'h_efff_0f17    // Hex literal, 32 bits wide (an AUIPC instruction)
'h23              // Hex literal, context determines width
```

When the size is omitted, *bsc* will infer the required size from the context, and extend it if necessary (zero-extend if the context requires a `Bit#(n)`, sign-extend if `Int#(n)`).

bsc will not truncate a too-large constant; instead it will give an error message.

Bit-vectors in BSV

- The basic type in any hardware design language is the bit-vector (a vector of n bits) to be treated as a single entity. Bit-vectors are carried on wires (n-bit vectors on n wires), stored in registers, memories and other state elements.
- The type of a bit-vector of n bits in BSV is written: `Bit#(`n`)`.
- We can declare identifiers with a type just like in Verilog, SystemVerilog and C, with an initialization:

```
1   Bit #(32) pc_val = ?;
2   Bit #(32) pc_val = 32'h_0000_1000;
3   Bit #(32) pc_val = 'h_1000;
```

Line 1: we let *bsc* pick an initial value (usually picks `'h_AAAA_..._AAAA` for visibilitye during debugging).

Line 2: the initial value is specified as an exactly 32-bit value, which matches the declared type of the identifier.

Line 3: the constant does not specify a width; *bsc* will infer that it should be 32 bits, and will zero-extend the given 16 bits.

Extracting smaller bit-vectors ("slicing"), or individual bits, from a bit-vector

```
Bit #(12) page_offset = pc_val [11:0];
Bit #(1)  pc_lsb      = pc_val [0];
Bit #(1)  pc_msb      = pc_val [31];
```

bsc checks that the bit-widths match exactly and reports an error otherwise.
(there is no silent bit-extending or truncating).

Operators for bit-vectors

Left- and right-arguments must have same `Bit#(n)` type.

Comparison ops: result type is `Bool`

```
if (a == b) ...;        // equality
if (a != b) ...;        // not-equal to
if (a < b) ...;         // less-than
if (a <= b) ...;        // less-than-or-equal-to
if (a > b) ...;         // greather-than
if (a >= b) ...;        // greater-than-or-equal-to
```

Arithmetic ops: result type is same as argument types

```
x = a + b - c * d;      // add, subtract, multiply
```

Bitwise logic ops: result type is same as argument types:

```
//   AND  OR   unary INVERT   XOR  XNOR  XNOR
x = a &  b |     (~ c)         ^   d ^~ e ~^ f;
```

Left- and Right-Shifts:

```
x = (a << 3) & (b >> 14);
```

Integer types in BSV

```
Bit #(n)        // bit-vectors, bounded to n bits
Int #(n)        // signed integers, bounded to n bits
UInt #(n)       // unsigned integers, bounded to n bits
Integer         // Mathematical integers (unbounded, no bit-width limit)
```

- We rarely use `UInt#(n)` because they are the same as `Bit#(n)` ("isomorphic").
- `Integer` is used for values that are only meaningful at compile time and never represented in hardware (such as the size of a vector of interfaces or modules).

Explicit extension and truncation

```
y = zeroExtend (x);
y = signExtend (x);
y = extend (x);
x = truncate (y);
```

- `x` and `y` must both be `Bit#(..)` or both be `Int#(..)`
- Bit-width of `y` must be $\geq$ Bit-width of `x`
- `extend` will zero-extend for `Bit#(..)` and sign-extend for `Int#(..)`