Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L4: **BSV**: Combinational Circuits

Reminders

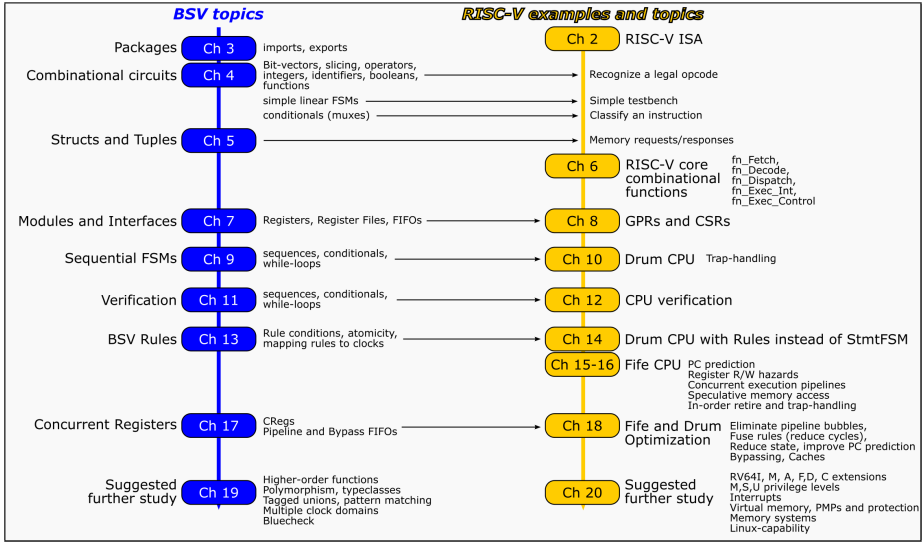
Please git clone or git pull: https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design

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

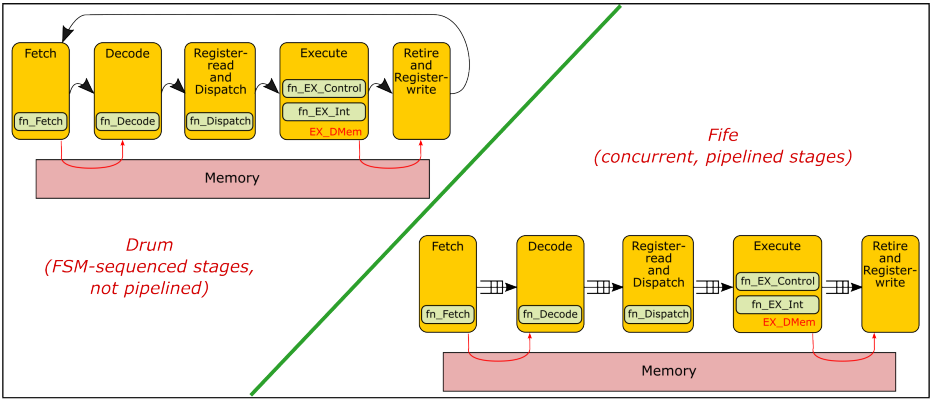
To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- *bsc* compiler (see https://github.com/B-Lang-org/bsc)
- Verilator compiler (see https://www.verilator.org/)

Chapter Roadmap



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 (see diagram on previous slide) 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 carry multiple pieces of data together.

Example: a memory request will carry a request-code (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:

- 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.

To compute these values, we will need to extract "slices" of the 32-bit instruction (opcode, funct3, rs1, rs2, rd, ...) and compare them with binary constants.

BSV: Integer literals (constants)

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)`).

BSV: Identifiers and comments

Identifiers: any sequence of alphabets, digits, and "_" (underscore) characters, beginning with an alphabet (same as in most programming languages):

The upper/lower case of the first letter (always an alphabet) is important:

- Uppercase first letter: constants (value constants, type constants).

  Examples:

  - Value constants: `True`, `False`, `MEM_RSP_OK`, ...
  - Type constants: `Bit`, `Int`, `Tuple2`, `Vector`, ...

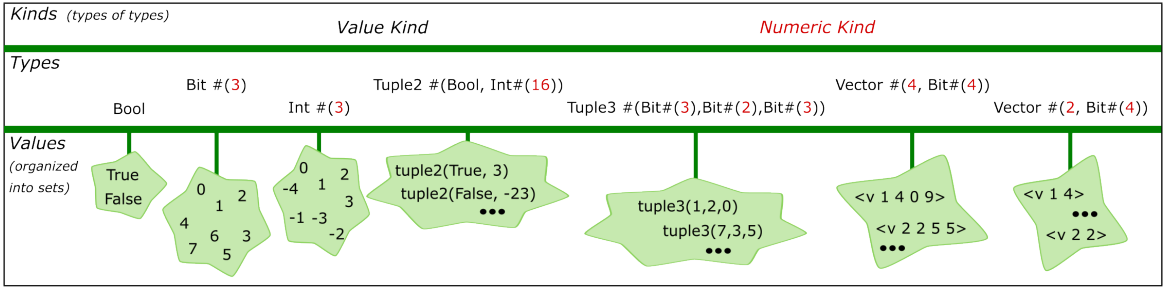
- Lowercase first letter: variables (value variables, type variables).

  Examples: `x`, `y`, `tmp`, `pc`, `rg_pc`, `has_rs1`, ...

Comments: same as in Verilog/SystemVerilog/C/C++:

- "//" introduces a comment until end-of-line
- "/*" and "*/" bracket an unlimited amount of comment text (can span multiple lines)

BSV: Introduction to Types



- Programs (and hardware modules) compute with *Values*.
- We group values into sets, which we call *Types*.
- Types themselves have a "type" (*Kind*):
  - those representing actual values (*Value Kind*)
  - those that describe some "size" feature of a type (*Numeric Kind*, shown in red)

Note: the numeric type "3" (shown in red) is distinct from the numeric value "3" (shown in black). There is never any ambiguity because they occur in distinct contexts: type expressions *vs.* value expressions.

BSV: Introduction to Types

BSV has very strong *type-checking*: every operator, function and method declaration in **BSV** specifies the types of its arguments and results, and these are checked strictly by *bsc*.

Every expression, statement, rule, module, ... in **BSV** is described by a *type expression* (or just "type" for short). Types can nested to arbitrary depth:

type ::= *type-constructor* `#(` *type*, ..., *type*)

A *type-constructor* always begins with an upper-case letter (is a type constant).

For each *type-constructor*, each *type* argument (parameter) is fixed to be either of value kind or numeric kind.
For example,

- In `Bit #(n)`, `n` always has numeric kind.
- In `Vector #(n,t)`, `n` always has numeric kind, `t` always has value kind.
- In `Tuple3 #(t1,t2,t3)`, all three parameters always have value kind.

BSV: Bit-vectors and declaring identifiers

- 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_8000_0000;
3   Bit #(32) pc_val = 'h_1000;
```

Line 1: we let *bsc* pick an initial value (usually picks `'h_AAAA_..._AAAA` to stand out during debugging).
Note: **BSV** does not have any Verilog-like concept of "X" values.

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

Exercise break

Please see directory: `Exercises/Ex_04_A_Bit_Vectors/`
and its README.

BSV: 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).

BSV: "let" bindings

When declaring an identifier, instead of specifying a type for the identifier, we can use the keyword "let" and leave it to the *bsc* compiler to infer the type:

```
let pc_val = 32'h_8000_0000;    // pc_val's inferred type: Bit #(32)
let pc_msb = pc_val [31];       // pc_msb's inferred type: Bit #(1)
```

This mechanism should only be used where the type is obvious for the human reader.

Exercise break

Please see directory: `Exercises/Ex_04_B_Bit_Vectors_Slicing/`
and its README

BSV: Operators on 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) ...;         // greater-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);
```

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#(..)`

BSV: the `Bool` type

`Bool`: the type of a boolean values, written `True` and `False`.

Operators

- `&&` (boolean/logical AND)
- `||` (boolean/logical OR)
- `!` (boolean/logical NOT)

CAUTION:

> `Bool`, `Bit#(1)` and `Int#(1)` are distinct types, and cannot be mixed!
>
> The boolean/logical operators `&&`, `||` and `!` operate on `Bool` types and are distinct from the bit-wise logic operators mentioned earlier (such as `&`), which operate on `Bit#(n)` types.

Bitwise comparison operators, such as `(a <= b)` take `Bit#(n)` arguments and produce `Bool` results.

BSV: Integer types

These are the two main integer types that we use in **BSV**:

```
Bit #(n)        // bit-vectors (unsigned integers), represented with n bits
Int #(n)        // signed integers, represented with n bits
```

Note: there are two more basic integer types available in **BSV**, that are less frequently used (by this author).

```
UInt #(n)       // unsigned integers, represented with n bits
```

We rarely use `UInt#(n)` because they are basically like `Bit#(n)` (same operators).

```
Integer         // Mathematical integers (unbounded, no bit-width limit)
```

`Integer` is used for values that are only meaningful at compile time and never represented in hardware (such as verbosity level for debugging or the size of a vector of interfaces).

BSV: User-defined functions

Syntax of function declarations is conventional (similar to Verilog, SystemVerilog, C):

```
function Action print_BV_BV_Bool (String op, Bit #(4) a, Bit #(4) b, Bool result);
   $display ("  %s: %04b %04b => %d or ", op, a, b, result, fshow (result));
endfunction
```

Syntax of function application is conventional (similar to Verilog, SystemVerilog, C):

```
      ...
      print_BV_BV_Bool ("==", a, b, a == b);
      ...
```

In this example, the result type is `Action`. This is used for functions that are pure side-effects: they perform some action and don't return any value.

Exercise break

Please see directory: `Exercises/Ex_04_C_Bit_Vectors_Operations/`
and its README.

BSV: User-defined functions have zero incremental hardware cost

In software, functions have some "function-calling overhead" because they perform some actions dynamically (allocate/deallocate stack frame, save/restore registers, move values to and from argument and result registers, ...).

In **BSV** functions are *inlined* wherever they are used, so there is no incremental hardware cost.

Takeaway: use functions liberally, to improve clarity, readability, reusability.

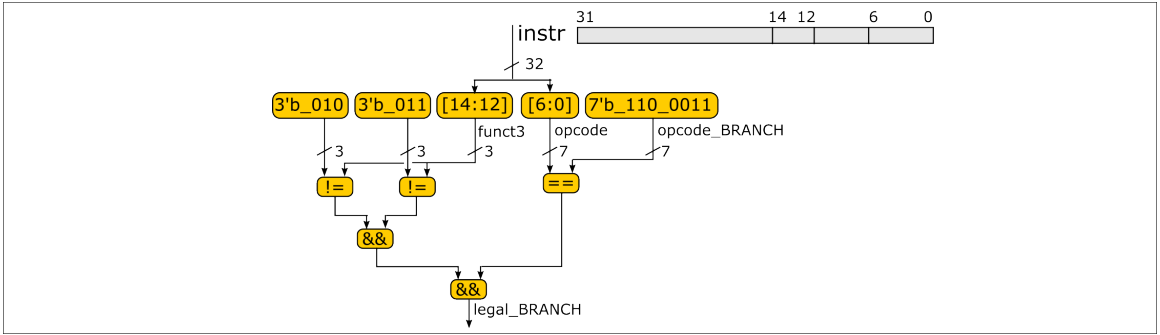
Example: recognizing a legal BRANCH insruction: code

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU

```
Bit #(32) instr         = ...;
Bit #(7)  opcode_BRANCH = 7'b_110_0011;

Bit #(7) opcode = instr [6:0];
Bit #(3) funct3 = instr [14:12];
Bool legal = (opcode == opcode_BRANCH)
             && (funct3 != 3'b010)
             && (funct3 != 3'b011));
```

Example: recognizing a legal BRANCH insruction: schematic



Note: the schematic is at "RTL" level; it does not go down to the level of AND-OR-NOT gates, just to bit-vector operators which will be implemented in terms of such gates by a synthesis tool.

Exercise break

Please see directory: `Exercises/Ex_04_D_is_legal_XXX/`
and its README.

Combinational circuits; pure *vs.* side-effecting functions; `Action` and `ActionValue` types

The function `is_legal_BRANCH()` is an example of a *combinational circuit*: an acyclic interconnection of primitive gates (such as AND, OR, NOT).

(More generally: interconnects of RTL-level binary operators on bit-vectors, since they are themselves combinational circuits).

- Combinational circuits do not have any *side-effects*—they do not modify any state elements. They are also said to be *pure* functions.
- We idealize combinational circuits as being "instantaneous" (zero time). In practice, because of physics, there will be a *propagation delay* for a change in an input signal to effect a change in the output, but as long as this is less than the clock period, we can regard it as instantaneous.
- Pure functions can be replicated or shared (un-replicated) without changing the functional meaning of the circut (replication/un-replication may have a non-functional implication: silicon size, combinational delay, power consumption, ...).
- In **BSV** circuits that may have a side-effect (may update a state element) always have type `Action` or `ActionValue#(t)`. Conversely, if a circuit's type does not involve `Action` or `ActionValue`, it is guaranteed to be pure.

Pure *vs.* side-effecting functions

Consider three versions of a function ...

Pure (no side-effect):

```
function Bit #(32)
         incr_pc_A (Bit #(32) pc);
   return pc + 4;
endfunction
```

Side-effecting (due to a $display) (and differing only in the text they print):

```
function ActionValue #(Bit #(32))
         incr_pc_B (Bit #(32) pc);
   actionvalue
      $display ("pc = %08h\n", pc);
      return pc + 4;
   endactionvalue
endfunction
```

```
function ActionValue #(Bit #(32))
         incr_pc_C (Bit #(32) pc);
   actionvalue
      $display ("The pc is %08h\n", pc);
      return pc + 4;
   endactionvalue
endfunction
```

The return-type of a function (or interface method) identifies it as:

pure:	`Bit#(32)`
vs. side-effecting:	`ActionValue#(Bit#(32))`

Invoking pure *vs.* side-effecting functions

All functions are invoked using traditional syntax (like C/C++, Verilog, SystemVerilog):

```
incr_pc_A (pc)
```

```
incr_pc_B (pc)
```

```
incr_pc_C (pc)
```

When a pure function is invoked, its result has *value type*, and so it can be bound to an indentifier using traditional syntax:

```
let new_pc = incr_pc_A (pc);
```

When an ActionValue function is invoked, its result has `ActionValue` type—it represents an action that *can be performed*. To actually perform the action, we use the following notation:

```
let new_pc <- incr_pc_B (pc);
```

[Advanced: "ActionValues are first-class types"] The reason for this difference is to allow the actionvalue itself to be treated as a value and bound to an identifier:

```
ActionValue #(Bit#(32)) a1 = incr_pc_B (pc);
ActionValue #(Bit#(32)) a2 = incr_pc_C (pc);
...
let new_pc <- (b ? a1 : a2);    // perform the action a1 or the action a2
```

This capability is common in modern functional programming languages (*e.g.*, Haskell).

Pragmatics: Wrapping a pure function as an `Actionvalue`

For a large/complex pure function, we sometimes deliberatly define it as an `ActionValue` type:

```
function Bit #(32)
         incr_pc_A (Bit #(32) pc);
   return pc + 4;
endfunction
```

$\Longrightarrow$

```
function ActionValue #(Bit #(32))
         incr_pc_B (Bit #(32) pc);
   actionvalue
      $display ("pc = %08h\n", pc);
      return pc + 4;
   endactionvalue
endfunction
```

in anticipation of the possibility that we may need to debug it later, because this makes it easy to insert a `$display()` (a side-effect).

We can always remove the `ActionValue` type later.

StmtFSM: a useful facility for testbenches

Many simple testbenches just involve performing a sequence of actions (providing stimulus/input to the DUT (Design Under Test). This is conveniently expressed using the following **BSV** idiom:

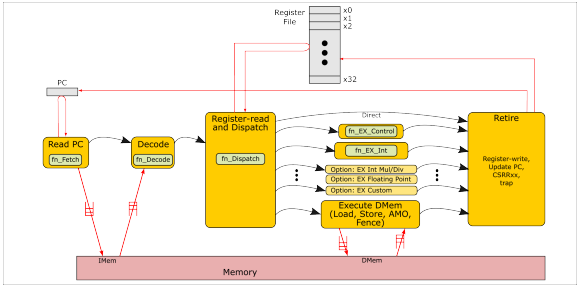
```
import StmtFSM ::  *
...
module mkTop (Empty);
  mkAutoFSM (
    seq
      action_1
      ...
      action_N
    endseq);
endmodule
```

We will discuss StmtFSM in more detail later (when we talk about the Drum CPU).
For now, just use the above idiom as-is.

Exercise break

Please see directory: `Exercises/Ex_04_E_FSM_Testbench/`
and its README.

User-defined types: `enum` types



In the "execute" stage, we have several alternative paths:

- Direct (for SYSTEM instructions)
- Control
- Integer arithmetic and logic
- Memory

The Decode stage computes a code that indicates which path should be taken.
The code is defined as an `enum` type:

src_Common/Inter_Stage.bsv: line 39 ...

```
typedef enum {OPCLASS_SYSTEM,     // EBREAK, ECALL, CSRRxx
              OPCLASS_CONTROL,    // BRANCH, JAL, JALR
              OPCLASS_INT,
              OPCLASS_MEM,        // LOAD, STORE, AMO
              OPCLASS_FENCE}      // FENCE
OpClass
deriving (Bits, Eq, FShow);
```

User-defined types: enum types

src_Common/Inter_Stage.bsv: line 39 ...

```
typedef enum {OPCLASS_SYSTEM,     // EBREAK, ECALL, CSRRxx
              OPCLASS_CONTROL,    // BRANCH, JAL, JALR
              OPCLASS_INT,
              OPCLASS_MEM,        // LOAD, STORE, AMO
              OPCLASS_FENCE}      // FENCE
OpClass
deriving (Bits, Eq, FShow);
```

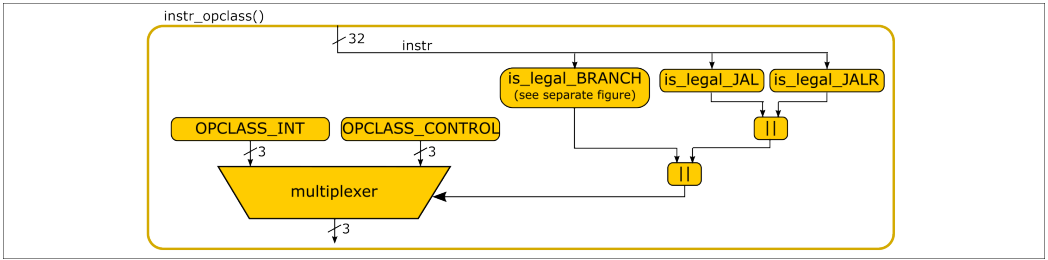
- These are symbolic (more human-readable) constants for the alternatives
- Because of "`deriving (Bits)`" *bsc* will represent them in 3 bits (`3'b_000` ... `3'b_100`)[2]
- However, `OpClass` is a new type, distinct from `Bit#(3)`
- You can use "`pack (OPCLASS_MEM)`" if you really want its `Bit#(3)` representation (`3'b_011`)

- Because of "`deriving (Eq)`" you can directly compare two `OpClass` values for equality ("`==`") and inequality ("`!=`")
- Because of "`deriving (FShow)`" you use "`fshow()`" on an `OpClass` value in `$display()` statements to print the symbolic name (otherwise, it will print the `Bit#(3)` representation)

[2]Without "`deriving`" you can give it a custom bit representation.

If-then-else (hardware multiplexers)

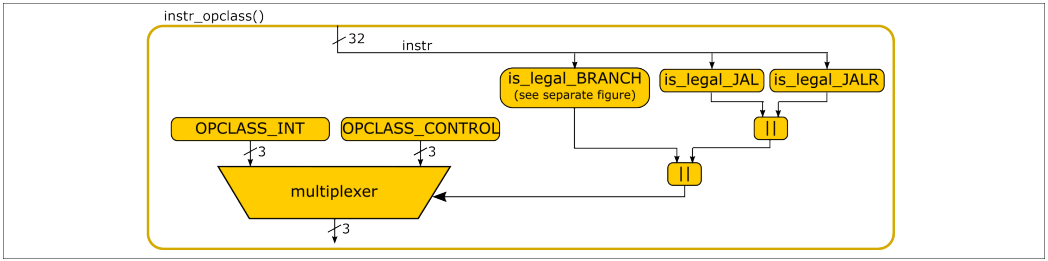
```
function OpClass instr_opclass (Bit #(32) instr);
   OpClass result;
   if (is_legal_BRANCH (instr) || is_legal_JAL (instr) || is_legal_JALR (instr))
      result = OPCLASS_CONTROL;
   else
      result = OPCLASS_INT;
   return result;
endfunction
```



Alternative notations for if-then-else

Conditional expressions:

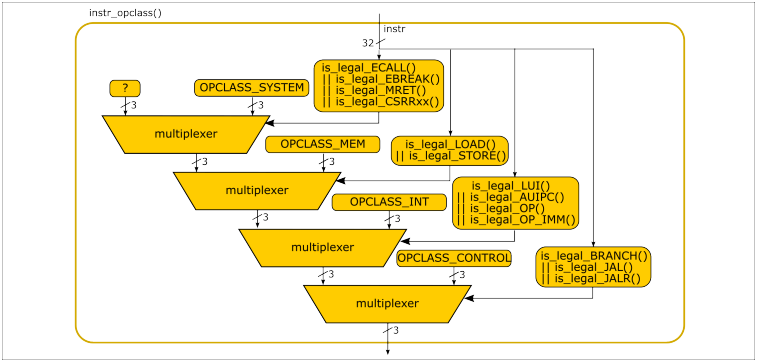
```
function OpClass instr_opclass (Bit #(32) instr);
   return ((is_legal_BRANCH (instr) || is_legal_JAL (instr) || is_legal_JALR (instr))
           ? OPCLASS_CONTROL
           : OPCLASS_INT);
endfunction
```



See also "case-endcase" expressions in the book and **BSV** Reference Guide.

Nested conditionals ⇒ cascaded multiplexers

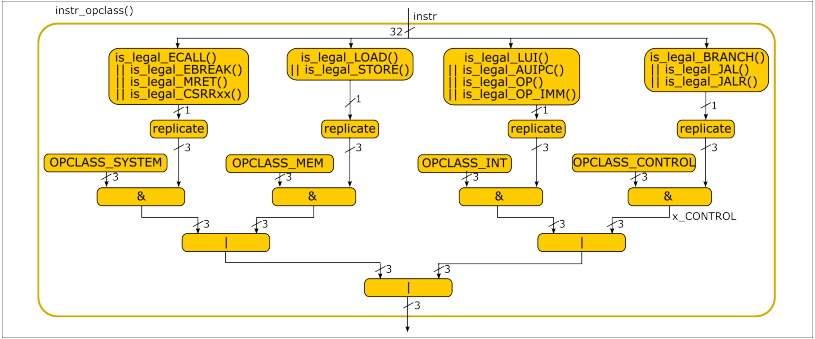
```
function Bool instr_opclass (Bit #(32) instr);
   OpClass result;
   if (is_legal_BRANCH (instr)
       || is_legal_JAL (instr)
       || is_legal_JALR (instr))
      result = OPCLASS_CONTROL;
   else if (is_legal_OP (instr)
            || is_legal_OP_IMM (instr)
            || is_legal_LUI (instr)
            || is_legal_AUIPC (instr))
      result = OPCLASS_INT;
   else if (is_legal_LOAD (instr)
            || is_legal_STORE (instr))
      result = OPCLASS_MEM;
   else if (is_legal_ECALL (instr)
            || is_legal_EBREAK (instr)
            || is_legal_MRET (instr)
            || is_legal_CSRRxx (instr))
      result = OPCLASS_SYSTEM;
   return result;
endfunction
```



Parallel muxes (AND-OR muxes, balanced muxes)

Cascaded multiplexers form an "unbalanced tree". We can balance the tree for a multiplexer with shorter combinational paths.

Note: this relies on the conditions being mutually exclusive and complete (exactly one of them is true):



Exercise break

Please see directory: `Exercises/Ex_04_F_Enums_Muxes/`
and its README.

Sharing code for RV32I and RV64I using type synonyms and macros

```
// type synonym: new name for numeric type 32
typedef 32 XLEN;

Bit #(XLEN) pc_val;
Bit #(XLEN) rs1_val;
Bit #(XLEN) rs2_val
Bit #(XLEN) rd_val;
```

Edit 32 → 64 for RV64

The following can automate the typedef of XLEN during compilation:

in src_Common/Arch.bsv

```
`ifdef RV32

typedef 32 XLEN;

`elsif RV64

typedef 64 XLEN;

`endif

Integer xlen = valueOf (XLEN);
```

Conditional compilation with values instead of ‘ifdef

For SLLI, SRLI and SRAI instructions, the “shift amount” (shamt):

- is 5 bits (instr[24:20]) in RV32I, and instr[25] must be 0
- is 6 bits (instr[25:20]) in RV64I, and instr[25] can be 0 or 1

If instr[25] is 1, it is illegal in RV32I. We can use xlen to test this in the decode function.

in src_Common/Instr_Bits.bsv

```
function Bool is_legal_OP_IMM (Bit #(32) instr);
   let funct3 = instr_funct3 (instr);
   let funct7 = instr_funct7 (instr);
   Bool is_legal_SLLI = (((xlen == 32) && (funct7 == 7'b000_0000))
                         || ((xlen == 64) && (funct7 [6:1] == 6'b0)));
   Bool is_legal_SRxI = ((   (xlen == 32) && ((funct7 == 7'b010_0000)
                                              || (funct7 == 7'b000_0000)))
                         || ((xlen == 64) && ((funct7 [6:1] == 6'b01_0000)
                                              || (funct7 [6:1] == 6'b00_0000))));
   return ((instr_opcode (instr) == opcode_OP_IMM)
           && ((funct3 == funct3_SLLI)
              ? is_legal_SLLI
               : ((funct3 == funct3_SRxI)
                  ? is_legal_SRxI
                  : True)));
endfunction
```

Conditional compilation with values instead of `ifdef: zero cost

Conditional compilation with values instead of `ifdef is preferable for readability as well as avoiding well known problems with macros ((scoping, inadvertant variable capture, inadvertant surprises due to associativity of infix operators, and so on)).

But is there a hardware cost (multiplexer for conditional)?

No, because an expression like "`xlen==32`" can, and is, statically evaluated to `True` or `False` by *bsc*, and the whole conditional is reduced to just the relevant arm (the conditional disappears).

End