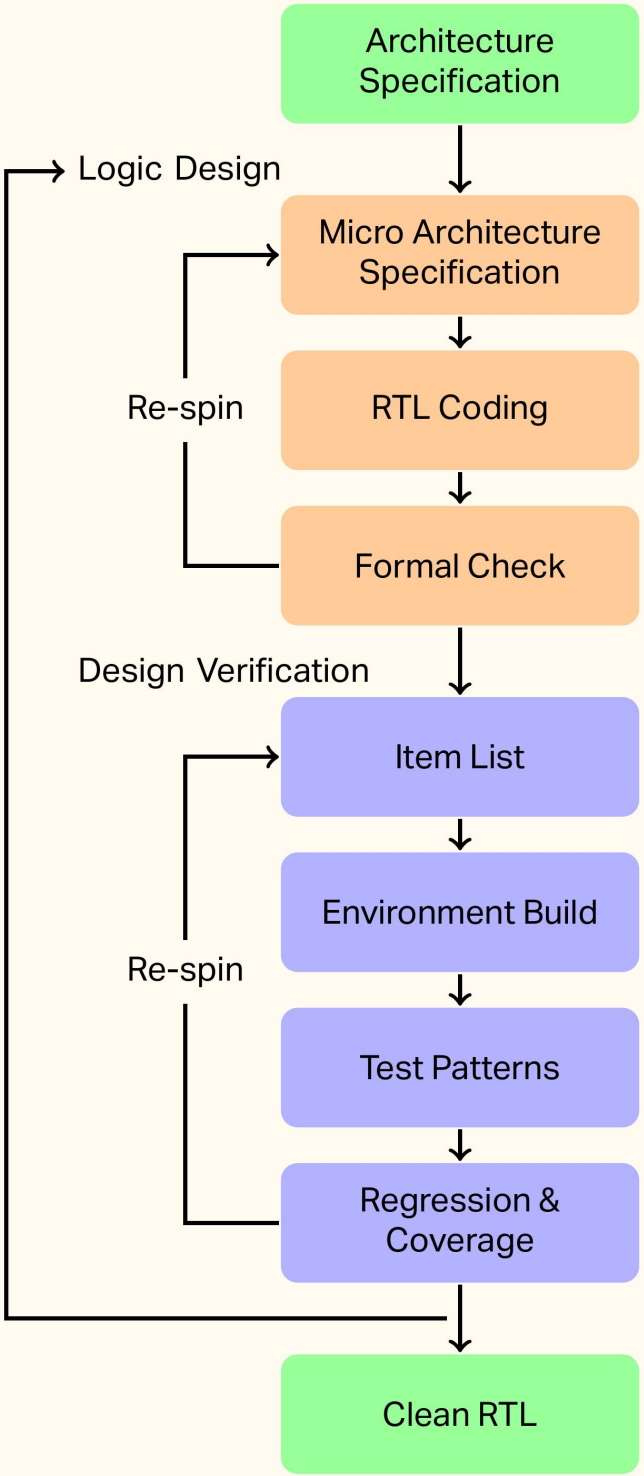
SystemVerilog 101

Presenter: Hai Cao Xuan
Computer Architecture 203B3



Describe detail of functional operation, detail implementation of each IP/module

Describe above specifications by using behavorial language such as SystemVerilog, SystemC, etc.

Test basic functions with some formal verification tools

Define check point to make sure design works as expected based on specifications

Build the testing environment where the design will be attached and run test sequences

Writing the test pattern that execute on the above EVN to examine the design behavior

Running massive verification invoking all test pattern with all randomized conditions. Sufficiency of test ensured by coverage database.

objectives

- Understand basics of SystemVerilog
- Understand how to model combinational logic
- Understand how to use always_comb and always_ff
- Undrestand the difference between blocking and non-blocking assignments
- Understand how to analyze FSMs
- Understand how to model basic components: Counter, Regfile, RAM
- Apply the coding guideline when coding

Gate-level Combinational Modeling

modules

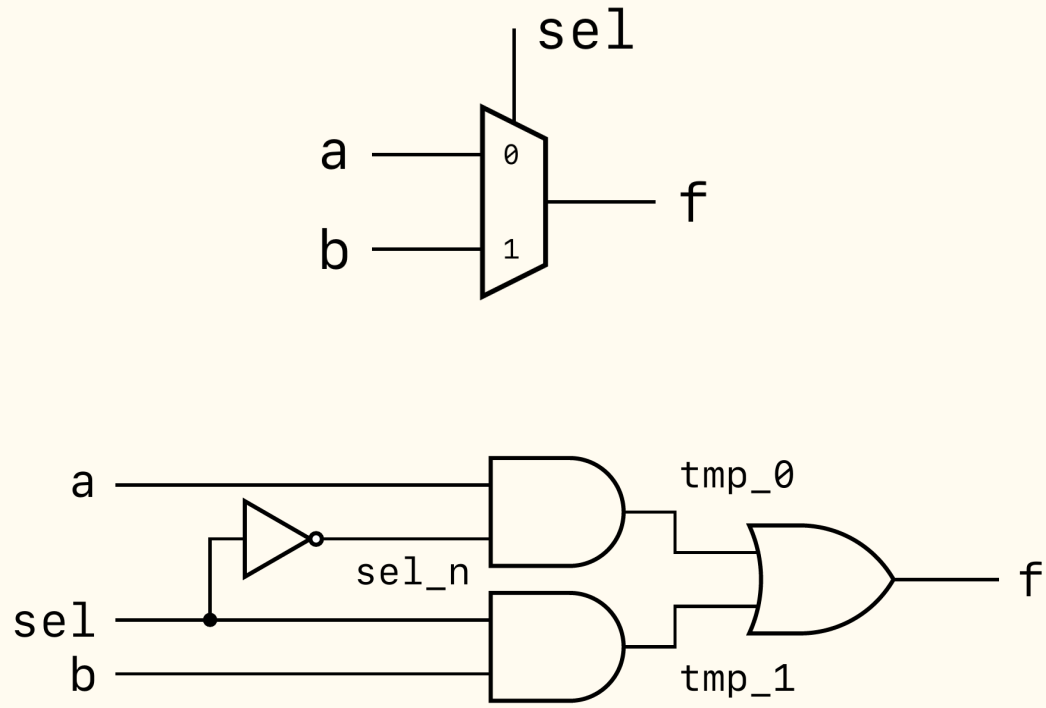
Module is the basic building block in SystemVerilog.

Every module usually defines:

- Input ports → Input signals of the circuit
- Output ports → Output signals of the circuit
- Concurrent statements → Logic of the circuit

modules

Exercise 1a. 2-1 Multiplexer



Netlist

Yosys

To view the schematic of your design, create a *.ys file, then run:

```
$ yosys file.ys
$ netlistsvg -o file.svg file.json
```

An *.svg will be in the folder, you could open it using your browser.

```
$ brave file.svg
```

modules

Logic data type

Use logic instead of wire and reg: **`0, 1, x, z`**

Vectors

Signals can be declared as vectors using **`[N:0]`** notation.

For example:

- an 8-bit vector → **`logic [7:0]`**
- an N-bit vector → **`logic [N-1:0]`**

Access the ith bit with **`[i]`**, or a bit slice from jth to ith with **`[i:j]`**

modules

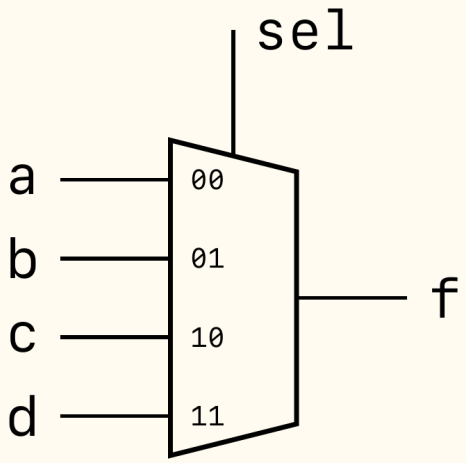
Module instantiations

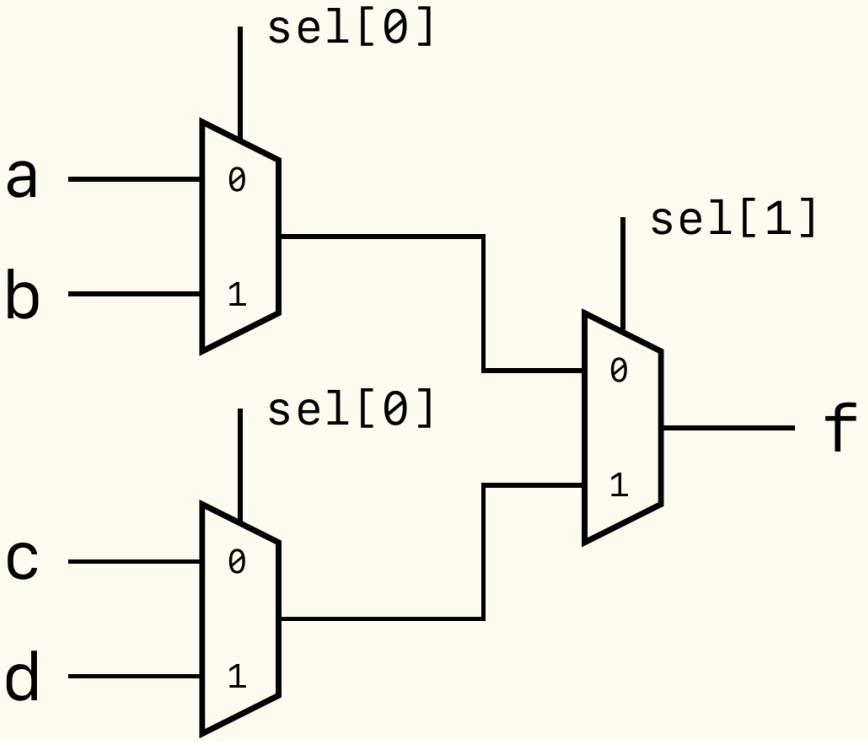
Reusing other modules in a “big” module is frequent.

→ See the example to know how to use this.

modules

Exercise 1b. 4-1 Multiplexer





Guideline

→ *Naming*

Declarations	**`lower_snake_case`**
Instance names	**`lower_snake_case`**
Signals (nets or ports)	**`lower_snake_case`**

→ *Suffixes for signal names*

Input	**`signal_i`**
Output	**`signal_o`**
Inout	**`signal_io`**
Active low	**`signal_n`**
Active high (if active low is present)	**`signal_p`**
? Input active low	**`signal_ni`**

literals

Binary → b: 0, 1

Decimal → d: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Hexadecimal → h: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

A literal may contain x or z.

A literal could be signed or unsigned, using “s”

literals

```
16'd8                → Decimal      → Unsigned → 16 bits
16'sd9               → Decimal      → Signed   → 16 bits
4'b1011              → Binary       → Unsigned → 4 bits
10'h2FA              → Hexadecimal  → Unsigned → 10 bits
12'b0110_0010_1101   → Use underscores for more readable
'0  → depend on the width of signal → N'b00...00
'1  → depend on the width of signal → N'b11...11
```

RT-level Combinational Modeling

continuous assignments

`assign` binds an expression (right-hand) to a signal (left-hand).

```
assign [signal] = [expression]
```

```
and (c_o, a_i, b_i) → assign c_o = a_i & b_i
```

Continuous assignments are continuous, so these two are equal:

```
logic [2:0] a;
logic       b;
logic       tmp;
assign tmp = a[0] | a[1];
assign b = tmp ^ a[2];
```

```
logic [2:0] a;
logic       b;
logic       tmp;
assign b = tmp ^ a[2];
assign tmp = a[0] | a[1];
```

continuous assignments

Bitwise operators

```
assign c[0] = a[0] | b[0]; → assign c = a | b;
assign c[1] = a[1] | b[1];
```

and → **&**, or → **|**, xor → **^**, not → **~**

Reduction operators

```
assign b = a[0] & a[1] & a[2]; → assign b = &a;
```

and → **&**, or → **|**, xor → **^**, nand → **~&**, nor → **~|**, xnor → **~^**

Logical operators

Only produce true or false (1 bit) → Should use for scalar values/signals (true/false)

and → **&&**, or → **||**, not → **!**

continuous assignments

Shift operators

left logical → **<<**, right logical → **>>**, right arithmetic → **>>>**

Arithmetic operators

add → **+**, subtract → **−**,

multiply → *****, divide → **/**, modulus → **%**, power → ******

Conditional operator

```
mux mux (a_i, b_i, sel_i, c_o) → assign c_o = sel_i ? b_i : a_i;
```

condition **?** result_if_true **:** result_if_false

continuous assignments

Comparison operators

`equal` → **`==`**, `less than` → **`<`**, `less than or equal` → **`<=`**

`not equal` → **`!=`**, `greater than` → **`>`**, `greater than or equal` → **`>=`**

Concatenate and replicate operators

`{a,b}`	concatenate vector a with vector b
`{n{a,b}}`	concatenate vector a with vector b and replicate the result n times
→ `{n{a},b}`	replicate vector a n times and concatenate with vector b

parameter and localparam

parameter the constant could be modified when instantiating

localparam the constant is “local” to the module

```
module module_name #(
  parameter UpperCamelCase        = ...,
  parameter AnotherUpperCamelCase = ...
) (
  ...
);

  localparam ALL_CAPS         = ...;
  localparam ANOTHER_ALL_CAPS = ...;

...

endmodule : module_name
```

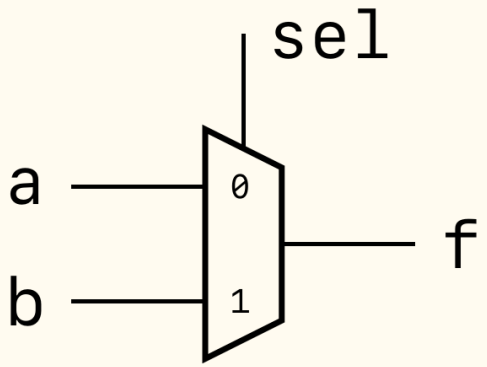
Guideline

→ *Naming*

Tunable Constants (parameter)	**`UpperCamelCase`**
True Constants (localparam)	**`ALL_CAPS`**

modules

Exercise 1c. 2-1 Multiplexer N-bit (configurable)



generate

generate helps create replicated structure.

```
generate
  genvar [index_variables];
  for ([initial]; [condition]; [inc/dec]) begin [: optional_label]
    [...concurrent_statements…]
  end
endgenerate
```

```
assign c[0] = a[0] | b[0];
assign c[1] = a[1] | b[1];
...
assign c[N] = a[N] | b[n];
```

→

```
generate
  genvar i;
  for (i = 0; i < N; i++) begin : gen_or
    assign c[i] = a[i] | b[i];
  end
endgenerate
```

always_comb

always_comb helps create combinational logics conveniently.
It is highly recommended than **always @(*)**

```
always_comb begin [: optional_label]
  [optional_local_variable_declarations]
  [...procedural_statements...]
end
```

This coding style is preferred.

```
logic [N-1:0] a;
always_comb begin : proc_example
   a = '0; // default value
   if (sel)
       a = b + c;
end
```

`always_comb` – blocking assignments

The value of the expression is evaluated and assigned to the variable/signal immediately

→ Next procedural statements will use that variable → Blocking

```
[variable_name] = [expression]; // blocking assignment
```

Should be used for always_comb only, aka combinational logic.

Compare these two

```
initial x = 0;
always_comb begin
  x <= 2;
  y <= x + 1;
end
```

```
initial x = 0;
always_comb begin
  y <= x + 1;
  x <= 2;
end
```

always_comb – if statements

```
if ([boolean]) begin
   [...procedural_statements...]
end else begin
   [...procedural_statements...]
end

if ([boolean]) begin
   [...procedural_statements...]
end
```

Remember **always assign default** values to signals/variables, especially when just one “if” is used.

always_comb – case statements

```
case [expression]
   [item]: begin
       [procedural_statements]
   end
   [item]: begin
       [procedural_statements]
   end
   ...
   default: begin
       [procedural_statements]
   end
endcase
```

Remember **always assign default** values to signals/variables, All cases should be considered.

Modeling Sequential Circuits

always

always uses a sensitivity list to perform value changes.

```
always @([sensitivity_lists]) begin [: optional_label]
  [optional_local_variable_declarations]
  [...procedural_statements...]
end
```

For example, sum will change if a or b change their values.

```
always @(a, b) begin : proc_example
   sum = a + b;
end
```

The sensitivity list uses the **posedge** or **negedge** keyword in front of a signal to specify the edge transition of that signal.

The wildcard ***** means all signals in all expressions of procedural statements are in the sensitivity list. → Look like another way to model combinational logic → **but should not use**

always – non-blocking assignments

The value of the expression is evaluated and assigned to the variable/signal at the end of always block.

→ Previous procedural statements will not block the following ones.

→ Not assign a value at least twice

→ Use non-blocking assignments to model sequential logic

```
[variable_name] <= [expression]; // non-blocking assignment
```

```
initial x = 0;
always @(posedge clk) begin
  x = 2;
  y = x + 1;
end
→ x = 2, y = 3
```

```
initial x = 0;
always @(posedge clk) begin
 y = x + 1;
 x = 2;
end
→ x = 2, y = 1
```

always_ff

always_ff notices the compiler the intention of synthesizing flip-flopsuses a sensitivity list to perform value changes.

→ Use this instead of plain always

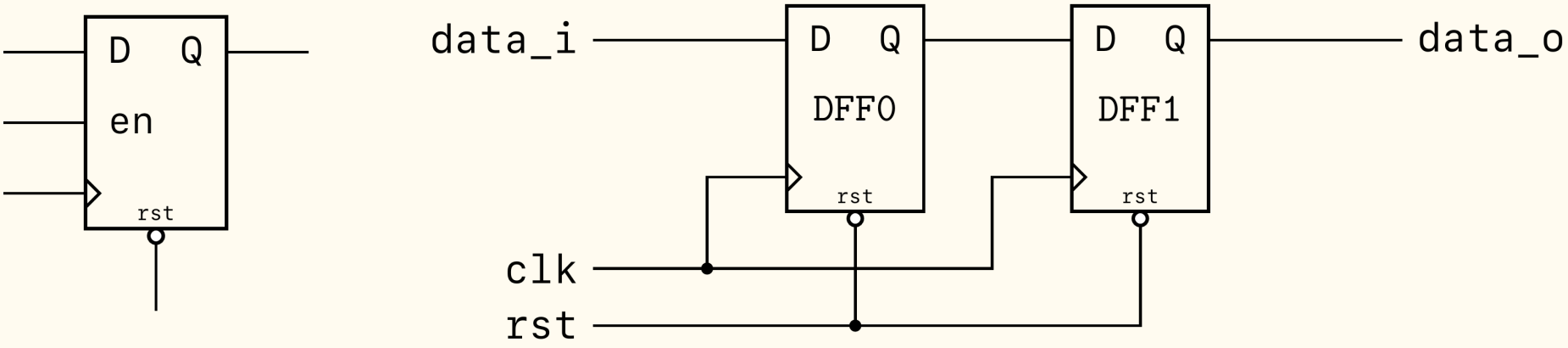
```
  always_ff @([sensitivity_lists]) begin [: optional_label]
    [optional_local_variable_declarations]
    [...procedural_statements...]
  end
```

```
always_ff @(posedge clk) begin : proc_example
   if (sel)
       a <= b + c;
end
```

modules

Exercise 2a. DFF with enable signal

Exercise 2b. 2-DFF Synchronizer



Guideline

→ Prefixes for optional label

generate	**`gen_`**
always...	**`proc_`**

sequential circuit

A general block diagram of a sequential logic contains:

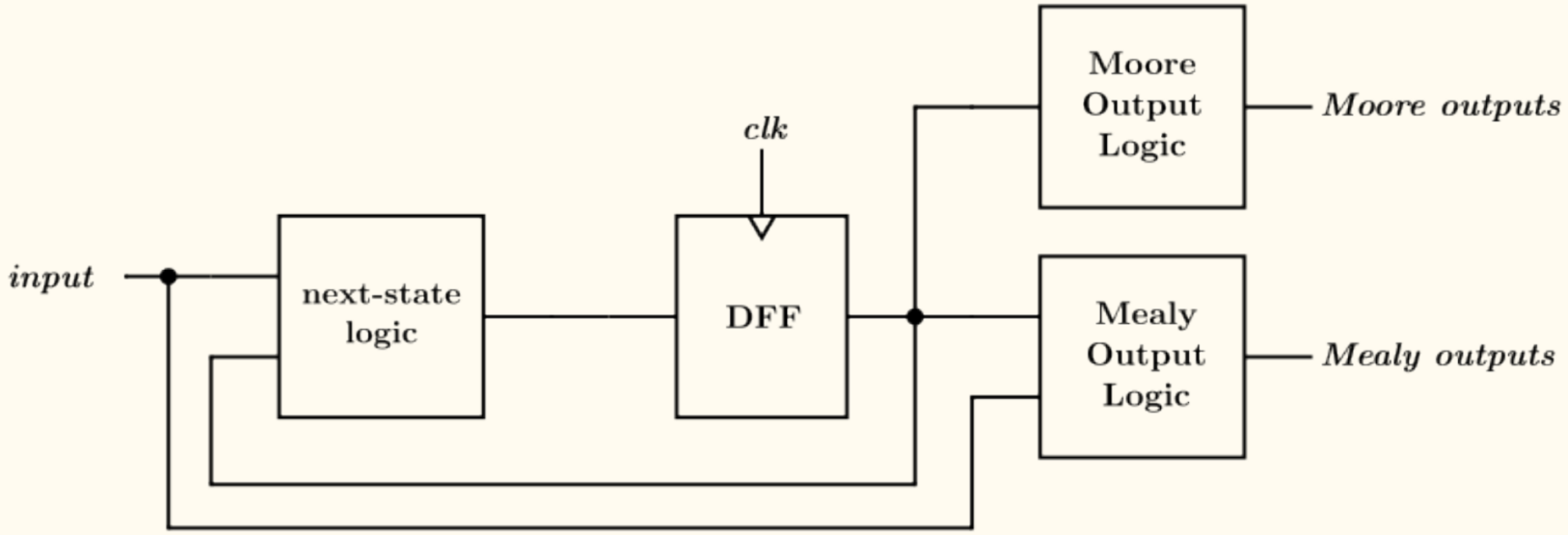
- State register → Flipflop
- Next-state logic → Combinational circuit
- Output logic → Combinational circuit

Modeling Finite State Machines

diagram and implementation

Moore machine → The output depends on the state only.

Mealy machine → The output depends on the state and the input.



enum – enumerations

If **logic** only has 4 states: 0, 1, x, z, **enum** helps create more.

typedef is to create user-defined data type.

```
typedef enum [storage_type]{
  [state_name] [= state_value,]
  [state_name] [= state_value,]
  ...
  [state_name] [= state_value]
} [new_name_type];
```

An FSM has three states: IDLE, PAUSE, ADD

```
typedef enum logic [1:0] {
   S_IDLE = 2'b00;
   S_PAUSE = 2'b01;
   S_ADD = 2'b10;
} state_e;
state_e current_state, next_state;
```

Guideline

→ *Suffix*

enum	**_e**

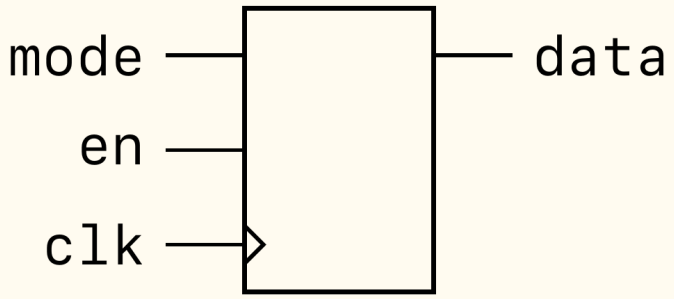
modules

Exercise 3. Universal counter

Two modes: Add and Subtract (by 1)

Counter only works when en = 1, or it will hold the current data.

Counter has an active low reset.



Modeling Memory

array

Packed Array

- Of single bit data types (logic), enumerated types, and even packed arrays

```
logic [7:0][3:0] array0;
state_e [2:0] state_array;
logic [3:0][7:0][3:0] packed_array;
```

- One dimensional packed array → vector → convenient to access a bit or bit slice

```
array0[2]; array0[3][5:3]; array0[1][1];
```

- Represented as a contiguous set of bits

array

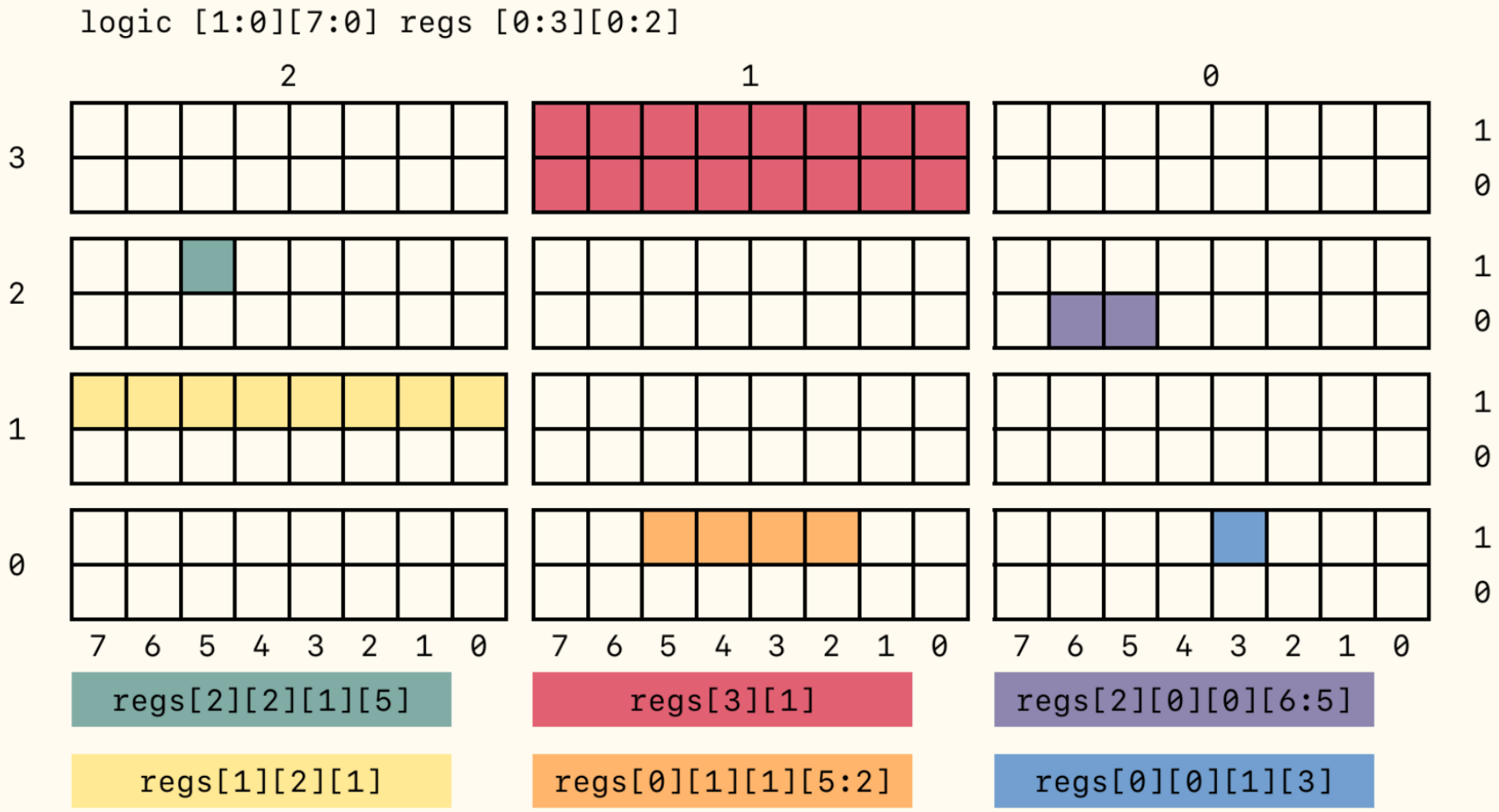
Unpacked Array

- Of any data types

```
logic uarray0[7:0][3:0];
state_e state_uarray[5:0];
logic [3:0][7:0] mix[3:0];
```

- Not guaranteed to be represented as a contiguous set of bits

array


logic [1:0][7:0] regs [0:3][0:2]
2
1
0
3
2
1
0
1
0
7 6 5 4 3 2 1 0
regs[2][2][1][5]
regs[3][1]
regs[2][0][0][6:5]
regs[1][2][1]
regs[0][1][1][5:2]
regs[0][0][1][3]

RAM and register file

Exercise 4a. Single-port RAM

Exercise 4b. Dual-port RAM

Exercise 4c. Regfile

A Little Bit Advanced

signal clusters

```
typedef struct packed{
  [storage_type] [signal];
  [storage_type] [signal];
  ...
  [storage_type] [signal];
} [cluster_type];
```

A module requires logic operator data and 2 operands. It could be configured as 1 data:

```
typedef struct packed{
   logic [2:0] logic_operator;
   logic [7:0] operand_a;
   logic [7:0] operand_b;
} logic_data_s;

logic_data_s data_in;

assign lu_op = data_in.logic_operator; // use “.” to access data
```

Guideline

→ *Suffix*

signal clusters	**_s**
other typedef	**_t**

package and *.svh files

When working on large projects, a number of parameters or types (enum, struct,…) will become global.

→ Painful if declaring them again and again in each module file

→ Modify one → Modify all

```
package [package_name];
  [declarations];
endpackage [: package_name]
```

→ Create *.svh files to store only packages

→ **`parameter`** will be declared in package → global

→ **`localparam`** will be in the specific module → local

Questions?

References

1. https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md
2. https://zyedidia.github.io/notes/sv_guide.pdf
3. http://courses.eees.dei.unibo.it/LABMPHSENG/wp-content/uploads/2016/02/SystemVerilog_3.1a.pdf