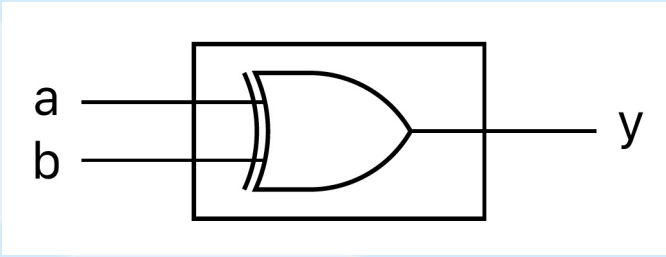
INTRODUCTION TO SYSTEM VERILOG

I. Combinational Logic Modeling – Mạch Tổ Hợp

1. Module và khai báo

❑ ***Module*** Một design (thiết kế) gồm có ba thành phần

- Ngõ vô (input)
- Ngõ ra (output)
- Mạch logic (logic circuit)



- Trong hình trên, có 2 input là **a** và **b**, 1 output là **y**, và mạch logic bên trong đơn giản chỉ là mạch **xor**. Mô tả mạch này trong SystemVerilog như sau:

```
1     module design_1 (
2       input logic  a, // ngõ vô a
3       input logic  b, // ngõ vô b
4
5       output logic y  // ngõ ra y
6     );
7
8       assign y = a ^ b; // mạch logic xor
9
10    endmodule : design_1
```

❑ ***Khai báo*** Để khai báo một signal (tín hiệu), ta cần gán cho nó data type (dạng dữ liệu), trong SystemVerilog, data type được sử dụng là **logic**.

- **logic** có 4 giá trị: **0, 1, x, z**. Data type này có thể tổng hợp được (synthesizable).
- Keyword **input** và **output** báo cho compiler biết signal này là port (cổng) của design.

2. Vector và Number

❑ ***Vector*** Trong thực tế, dữ liệu không phải lúc nào cũng chỉ có 1 bit (1 dây signal) mà có nhiều, ví dụ 1 byte = 8 bit, hay signal có 8 dây. Khai báo một vector như sau:

```
logic [3:0] four_bit_data;
logic [7:0] one_byte_data;
logic [31:0] four_byte_data;
```

❖ *Để giúp code trở nên dễ đọc và dễ debug về sau, khuyến khích khai báo tín hiệu kèm hậu tố* ***_i*** *hoặc* ***_o*** *tùy vô tín hiệu là* ***input*** *hoặc* ***output****.*

❖ *Tên* ***signal*** *và tên* ***module*** *nên viết dưới dạng* ***lower_snake_case***

```
1     module design_2 (
2       input logic [2:0]  a_i,
3       input logic [2:0]  b_i,
4
5       output logic       x_o,
6       output logic       y_o,
7       output logic [1:0] z_o
8     );
9
10      assign x_o = a[0] & b[1];     // mạch logic and
11      assign y_o = a[0] ^ b[0];     // mạch logic xor
12      assign z_o = a[1:0] | b[1:0]; // mạch logic or
13
14    endmodule : design_2
```

- Với một vector signal được khai báo như sau: logic [7:0] signal
  - Truy cập bit thứ i (0 ≤ i ≤ 7): signal[i]
  - Truy cập một slice bit từ bit thứ i tới bit thứ j (0 ≤ i ≤ j ≤ 7): signal[j:i]

2. Vector và Number

❑ ***Number*** SystemVerilog quy định number dựa theo 4 yếu tố: độ dài, hệ số, dấu toán học, và giá trị.

```
4'b1011                  : 4 bit unsigned binary
8'd35                    : 8 bit unsigned decimal
8'sd63                   : 8 bit signed decimal
16'h7F5C                 : 16 bit unsigned hexadecimal
16'b0001_1100_1010_0110 : use "_" for readability

logic [7:0] foo;     // declare an 8bit signal named foo
assign foo = 8'h3A; // assign a value of 3A (00111010) to foo
```

❖ *Lưu ý độ dài number là số bit hay xét theo binary, không phải số ký tự*

❖ *Khi gán giá trị, phải để ý độ dài bit đặng tránh khác nhau độ dài → bug*

○ Trong SystemVerilog còn có một dạng gán giá trị như sau:

```
logic [15:0] foo;    // declare a 15bit named foo
logic [7:0] bar;     // declare an 8 bit named bar

assign foo = '0; // assign 0 to all of the bits of foo
                 // the same as assign foo = 16'b0;
assign bar = '1; // assign 1 to all of the bits of bar
                 // the same as assign bar = 8'b11111111;
                 //          or assign bar = 8'hFF;
```

3. Assign

- **Assign** hay **continuous assignment**, tức gán liên tục, có thể hiểu như mạch tổ hợp ngoài thực tế, liên tục gán giá trị sau khi "tính toán" từ ngõ vô lên ngõ ra.

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

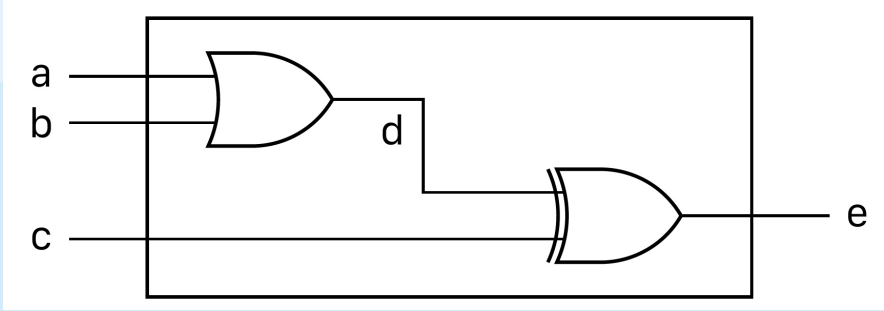
  - Như tên của nó, continuous assignment, nên vị trí các dòng assign không ảnh hưởng tới kết quả sau cùng.

```
logic data_a;
logic data_b;
logic data_c;
logic data_d;
logic data_e;

assign data_d = data_a | data_b;
assign data_e = data_d ^ data_c;
```

```
logic data_a;
logic data_b;
logic data_c;
logic data_d;
logic data_e;

assign data_e = data_d ^ data_c;
assign data_d = data_a | data_b;
```



3.1 Bitwise operators – Phép toán với bit

- Bốn phép toán Boolean cơ bản:

```
& ⇒ and          | ⇒ or
^ ⇒ xor          ~ ⇒ not
```

```
assign data_and = foo & bar;
assign data_or  = foo | bar;
assign data_xor = foo ^ bar;
assign data_not = ~foo;
```

- Đối với vector, từng cặp bit tương ứng sẽ thực hiện phép toán bitwise với nhau.

```
logic [3:0] foo;
logic [3:0] bar;
logic [3:0] qux;
logic [3:0] sal;

assign sal = foo ^ bar | qux;
```

```
For example:
foo = 4'b0110;
bar = 4'b1100;
qux = 4'b1010;
⇒ sal = 4'b1010
```

```
logic [3:0] foo;
logic [3:0] bar;
logic [3:0] qux;
logic [3:0] sal;

assign sal = foo ^ (bar | qux);
```

```
For example:
foo = 4'b0110;
bar = 4'b1100;
qux = 4'b1010;
⇒ sal = 4'b1010
```

3.2 Reduciton operators – Phép toán rút gọn bit

- SystemVerilog cung cấp Reduction operators cho những trường hợp cần lấy một bit từ một signal nhiều bit bằng một phép toán.

`&` ⇒ and `|` ⇒ or `^` ⇒ xor	`~&` ⇒ nand `~|` ⇒ nor `~^` ⇒ xnor

```
logic [3:0] foo;
logic       bar;

assign bar = foo[3] | foo[2] |
             foo[1] | foo[0];
```

```
logic [3:0] foo;
logic       bar;

assign bar = |foo;
```

3.3 Logic operators – Phép toán luận lý

- Logical operators gần giống với bitwise operators, tuy nhiên nó chỉ tính toán giá trị true/false.

`&&` ⇒ and	`||` ⇒ or	`!` ⇒ not

❖ *Nên sử dụng logical operators với giá trị đơn (1-bit), để xác định giá trị true/false (điều kiện).*

3.4 Comparison Operators – Phép toán so sánh

- Những phép toán so sánh sau là synthesizable:

```
==  ⇒ equal                 !=  ⇒ not equal
<   ⇒ less than             <=  ⇒ less than or equal
>   ⇒ greater than          >=  ⇒ greater than or equal
```

3.5 Shift operators – Phép toán dịch

- Có ba phép toán dịch cơ bản mà SystemVerilog cung cấp:

```
<<  ⇒ shift left logical
>>  ⇒ shift right logical
>>> ⇒ shift right arithmetic
```

```
assign shift_left = foo << bar;
assign shift_rigl = foo >> bar;
assign shift_riga = foo >> bar;
```

```
For example:
foo = 8'b1001_1010; bar = 8'b11; (or 8'd3)
⇒ shift_left = 8'b1101_0000
⇒ shift_rigl = 8'b0001_0011
⇒ shift_riga = 8'b1111_0011
```

3.6 Arithmetic operators – Phép toán số học

- Những phép toán trên phần nhiều là phép toán Boolean, ngoài ra phép toán số học cũng được SystemVerilog hỗ trợ.

+ ⇒ add	- ⇒ subtract
* ⇒ multiply	/ ⇒ divide
% ⇒ modulus	** ⇒ power

  - ❖ *Cần lưu ý đến việc sử dụng Arithmetic operators, ví dụ cần có FPGA có hỗ trợ bộ nhân mới nên sử dụng phép * khi **assign**.*
  - ❖ *Phép toán + và - thì hầu hết đều được hỗ trợ, nhưng các phép toán còn lại chỉ nên sử dụng để tính toán số học, ví dụ các thông số, thay vì dùng trong **assign**.*

3.7 Conditional operators – Phép toán điều kiện

- Phép toán điều kiện là cách ngắn gọn thay vì tạo một bộ mux và sử dụng nó. Phép toán này có dạng giống như trong C.

```
[condition] ? [result_if_true] : [result_if_false]
```

```
logic foo;
logic bar;
logic sel;
logic qux;

assign qux = (foo & sel) |
             (bar | ~sel);
```

```
logic foo;
logic bar;
logic sel;
logic qux;

assign qux = sel ? foo : bar;
```

❖ *Condition phải mang giá trị true/false, nên cần đặt condition là signal đơn (1-bit), reduction operators, logical operators, hoặc phép so sánh.*

3.8 Concatenate and replicate operators – Phép toán ghép và lặp số

- Đối với việc tạo ra vector có số bit lớn, hoặc ghép nhiều signal lại thành một signal có số bit lớn.

```
{foo, bar}  ⇒ concatenate
{N{foo}}    ⇒ replicate
```

```
logic [2:0]  foo;
logic [3:0]  bar;
logic [6:0]  data_a;
logic [9:0]  data_b;
logic [10:0] data_c;
logic [17:0] data_d;

assign data_a = {foo, bar};
assign data_b = {3{foo}};
assign data_c = {bar, {2{foo}}};
assign data_d = {data_a, data_b};
```

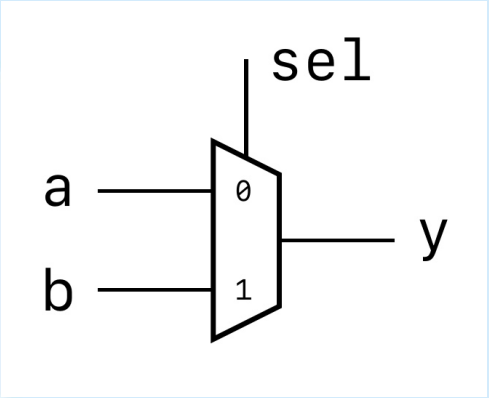
❖ *Cần lưu ý tới độ dài bit của signal khi concatenate hoặc replicate*

```
For example:
foo = 3'b101; bar = 4'b1100;
⇒ data_a =  7'b1011100; (101_1100)
⇒ data_b = 10'b101101101; (101_101_101)
⇒ data_c = 11'b1100101101; (1100_101_101)
⇒ data_d = 17'b10111001011101101; (1011100_101101101)
```

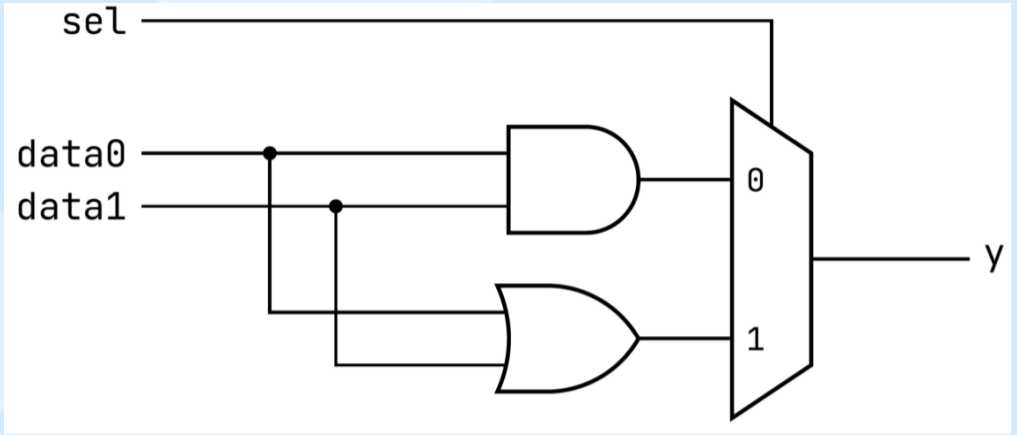
4. Instantiation

- Instantiation chỉ sự sử dụng một hay nhiều module khác đã tạo trong module hiện tại. Ví dụ:

```
1     module mux_two (
2       input logic [3:0]  a_i,
3       input logic [3:0]  b_i,
4       input logic        sel_i,
5       output logic [3:0] y_o
6     );
7
8       assign y_o = sel_i ? b_i : a_i;
9
10    endmodule : mux_two
```




```
1     module design_3 (
2       input logic [3:0]  data0_i,
3       input logic [3:0]  data1_i,
4       input logic        sel_i,
5       output logic [3:0] y_o
6     );
7
8       logic [3:0] and_tmp;
9       logic [3:0] xor_tmp;
10
11      assign and_tmp = data0_i & data1_i;
12      assign xor_tmp = data0_i ^ data1_i;
13
14      mux_two mux_two0 (
15        .a_i  (and_tmp),
16        .b_i  (xor_tmp),
17        .sel_i(sel_i  ),
18        .y_o  (y_o)
19      );
20
21    endmodule : design_3
```



4. Instantiation

- Có nhiều cách để khai báo/gán signals/data vô các ports của module, nhưng cách trên bảo đảm chúng được nối với nhau cách chính xác.

```
// signals/data wired to module's ports need to be in order
mux_two mux_two0 (and_tmp, xor_tmp, sel_i, y_o);

// or
mux_two mux_two0 (
  and_tmp,
  xor_tmp,
   sel_i,
   y_o
 );
```

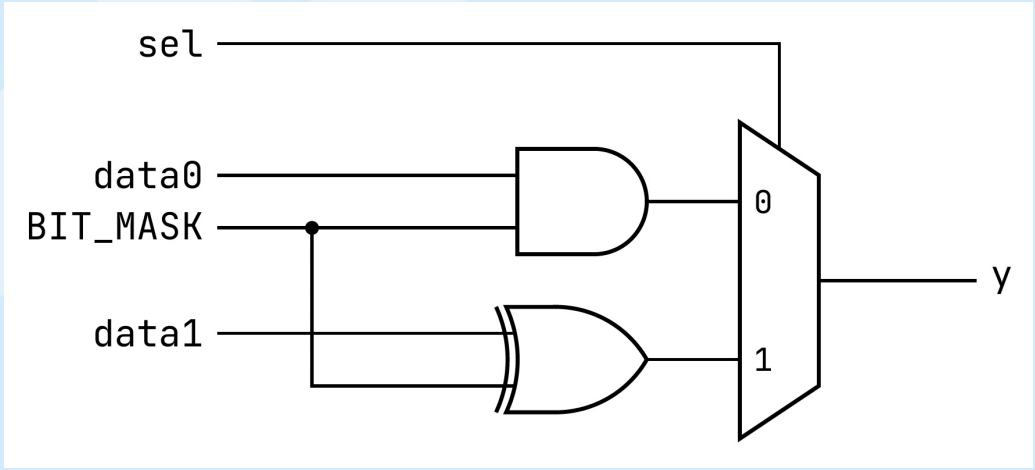
```
// when signals/data names are identical to ports' ones.
mux_two mux_two0 (
  .a_i  (and_tmp),
  .b_i  (xor_tmp),
  .sel_i,
  .y_o
);

// or
mux_two mux_two0 (
  .a_i  (and_tmp ),
  .b_i  (xor_tmp ),
  .*
);
```

5. Parameter và localparam

- Có hai cách để khai báo hằng số, **parameter** và **localparam**, điểm khác nhau ở chỗ **parameter** có thể được gán trong lúc sử dụng module (instantiation), còn **localparam** chỉ sử dụng nội trong module đó.

```
1     module mux_two_nbit # (
2       parameter DataSize = 4
3     ) (
4       input logic [DataSize-1:0]  a_i,
5       input logic [DataSize-1:0]  b_i,
6       input logic                 sel_i,
7       output logic [DataSize-1:0] y_o
8     );
9
10      assign y_o = sel_i ? b_i | a_i;
11
12    endmodule : mux_two_nbit
```



5. Parameter và localparam

```
1     module design_4 (
2       input logic [7:0]  data0_i,
3       input logic [7:0]  data1_i,
4       input logic        sel_i,
5       output logic [7:0] y_o
6     );
7
8       localparam BIT_MASK = 8'b11001100;
9
10      logic [7:0] and_tmp;
11      logic [7:0] xor_tmp;
12
13      assign and_tmp = data0_i & BIT_MASK;
14      assign xor_tmp = data1_i ^ BIT_MASK;
15
16      mux_two_nbit # (
17        .DataSize(8)
18      ) mux_two_nbit0 (
19        .a_i  (and_tmp),
20        .b_i  (xor_tmp),
21        .sel_i(sel_i  ),
22        .y_o  (y_o)
23      );
24
25    endmodule : design_4
```

- Mặc dù giá trị của DataSize được gán vốn là 4 trong **mux_two_nbit**, khi **design_4** sử dụng **mux_two_nbit**, nó thể thay đổi giá trị đó thành 8.

- *Tên **parameter** nên viết dưới dạng **UpperCamelCase***
- *Tên **localparam** nên viết dưới dạng **ALL_CAPS***

6. Always_comb

- Keyword này có thể được hiểu như việc thực hiện các statement liên tục. Nếu có nhiều phép gán, các statement cần đặt trong **begin** và **end**. Đặt tên cho **always_comb** là không bắt buộc nhưng nên làm vì sẽ dễ debug hơn.

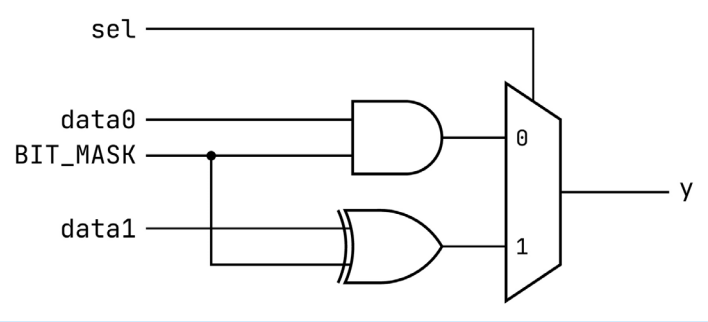
```
always_comb begin : proc_name
  ...
end
```

```
1   module design_4 (
2     input logic [7:0]  data0_i,
3     input logic [7:0]  data1_i,
4     input logic        sel_i,
5     output logic [7:0] y_o
6   );
7
8     localparam BIT_MASK = 8'b11001100;
9
10    logic [7:0] and_tmp;
11    logic [7:0] xor_tmp;
12
13    always_comb begin : proc_logic
14      and_tmp = data0_i & BIT_MASK;
15      xor_tmp = data1_i ^ BIT_MASK;
16      y_o = sel_i ? xor_tmp : and_tmp;
17    end
18
19  endmodule : design_4
```

- Bên trong **always_comb**, các phép gán này gọi là blocking statement, vì statement sau sẽ sử dụng kết quả của statement trước.

`initial foo = 0;`
`always_comb begin`
`  bar = foo + 1;`
`  foo = 2;`
`end`
	`initial foo = 0;`
	`always_comb begin`
	`  foo = 2;`
	`  bar = foo + 1;`
	`end`

After run ⇒ foo = 2; bar = 1	After run ⇒ foo = 2; bar = 3



- Ngoài blocking-statement, một vài synthesizable statements khác cũng có thể được sử dụng như: **if** và **case**.

❖ *Nên sử dụng **always_comb** khi thiết kế combinational logic phức tạp.*

❖ *Với bản chất của blocking statement, cần biết rõ thứ tự các khối logic để nối dây cách chính xác.*

7. If Statement

- Mạch mux có thể viết lại với if statement như sau:

```
always_comb begin : proc_mux
  if (sel_o == 1'b0)  // result of the expression inside
    y_o = a_i;        // the parentheses must be true or false
  else
    y_o = b_i;
end
```

```
always_comb begin : proc_mux
  if (sel_o == 1'b0) begin
    y_o = a_i;
  end
  else begin
    y_o = b_i;
  end
end
```

```
always_comb begin : proc_sample
  if (sel_o[1] == 1'b0) begin
    y_o = a_i;
  end
  else begin
    if (sel_o[0] == 1'b0) begin
      y_o = b_i;
    end
    else begin
      y_o = c_i;
    end
  end
end
```

8. Case Statement

- Nếu đã quen với **switch** trong C, **case** trong SystemVerilog cũng hoạt động tương tự.

```
always_comb begin : proc_sample
  case (sel_o)  // sel_o will be compared to each item
    2'b00: y_o = a_i;
    2'b01: y_o = b_i;
    2'b10: y_o = c_i;
    2'b11: begin
           y_o = d_i;
    end
  endcase
end
```

```
always_comb begin : proc_sample
  case (sel_o)  // sel_o will be compared to each item
    3'b000:  y_o = a_i;
    3'b001:  y_o = b_i;
    3'b100:  y_o = c_i;
    3'b111:  y_o = d_i;
    default: y_o = '0;  // assign '0 to y_o
  endcase               // if sel_o doesn't match other cases
end
```

❖ *Khi thiết kế cần bảo đảm sự đầy đủ của các giá trị **case** hoặc có **default**.*

9. Packed Array

- Một vector có thể hoạt động như một thanh ghi, ví dụ khai báo sau: **logic [31:0]** register; một thanh ghi 32 bit gồm có 4 byte, **[31:24], [23:16], [15:8], [7:0]**.
- Tuy nhiên, việc truy cập từng byte nhỏ như trên sẽ trở nên khó khăn nếu truy cập thường xuyên.
- Xét cơ bản, array cũng tương tự như vector nhưng giúp việc truy cập những bit bên trong có dễ hơn.

```
logic [31:0] register; ⇔ logic [3:0][7:0] register;
```

- Lúc này việc truy cập các byte sẽ tiện hơn rất nhiều.

```
register[23:16]; ⇔ register[2];
 register[7:0]; ⇔ register[0];
register[3:0]; ⇔ register[0][3:0];
```

- Nhìn từ trái qua phải, register khai báo ở dạng array được hiểu như sau:
- Có thể hiểu vector như một bảng chỉ có 1 hàng, còn array là một bảng có nhiều hàng, hoặc nhiều tập hợp bảng.

```
logic [3:0][7:0] register; ⇒ array named register
⇒ consist of 4 vector ⇒ 8 bit/vector
```

```
Access vector 2: register[2] (8 bit)
Access vector 0-1 : register[1:0] (16 bit)
Access bit 0-3 của vector 3: register[3][3:0] (4 bit)
```

```
logic [15:0][3:0][7:0] parray;  ⇒ array named parray
⇒ consist of 16 array ⇒ 4 vector/array ⇒ 8 bit/vector
```

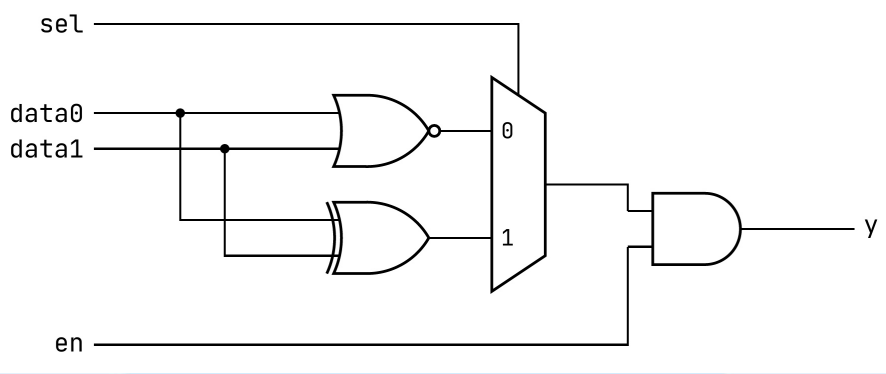
10. Unpacked Array

- Khi các vector không cần phải liên tiếp nhau như packed array, unpacked array cũng có thể được sử dụng.

```
logic [31:0] register [7:0];
```

11. Bài Tập

1. Thiết kế mạch combinational sau sử dụng **assign** và **always_comb**. (2 Ξles)



2. Thiết kế bộ tính toán đơn giản N bit với yêu cầu sau:
    a. 3 ngõ vào **data0**, **data1**, và **sel**
    b. 1 ngõ ra **result**
    c. 4 phép toán (NONE, ADD, AND, OR)

sel	result
00	0
01	data0 + data1
10	data0 & data1
11	data0 | data1

II. Sequential Logic Modeling – Mạch Tuần Tự

1. Always

- **always** tương tự như **always_comb** nhưng nó không thực hiện các statement liên tục mà dựa vô sự thay đổi của sensitivity list.

```
always @([sensitivity list]) begin : proc_name
  ...
end
```

```
always @(a, b) begin : proc_add
  sum = a + b; // sum thay đổi khi a hoặc b thay đổi
end
```

```
always @* begin : proc_add
  sum = a + b; // sum thay đổi khi a hoặc b thay đổi
end
```

```
always @(posedge a or b) begin : proc_add
  sum = a + b; // sum thay đổi khi a thay đổi 0 → 1
end            // hoặc b thay đổi
```

```
always @(negedge a or b) begin : proc_add
  sum = a + b; // sum thay đổi khi a thay đổi 1 → 0
end            // hoặc b thay đổi
```

- *Trường hợp thứ 1 và 2 có thể hoạt động giống **always_comb** nhưng không khuyến khích sử dụng thay thế **always_comb** vì những điểm không hoàn hảo mà trong đây không tiện đề cập.*

1. Always

- Ngoài blocking statement thì trong always còn có thể chứa non-blocking statement. Giả sử trước thời điểm xung clk kích cạnh lên, giá trị lúc đó của **foo** là 1, so sánh bốn kết quả sau sau khi xung clk kích cạnh lên.

blocking	**blocking**

`always @(posedge clk) begin`
`  bar = foo + 1;`
`  foo = 2;`
`end`
	`always @(posedge clk) begin`
	`  foo = 2;`
	`  bar = foo + 1;`
	`end`

After posedge clk ⇒ foo = 2; bar = 2	After posedge clk ⇒ foo = 2; bar = 3

non-blocking	**non-blocking**

`always @(posedge clk) begin`
`  bar <= foo + 1;`
`  foo <= 2;`
`end`
	`always @(posedge clk) begin`
	`  foo <= 2;`
	`  bar <= foo + 1;`
	`end`

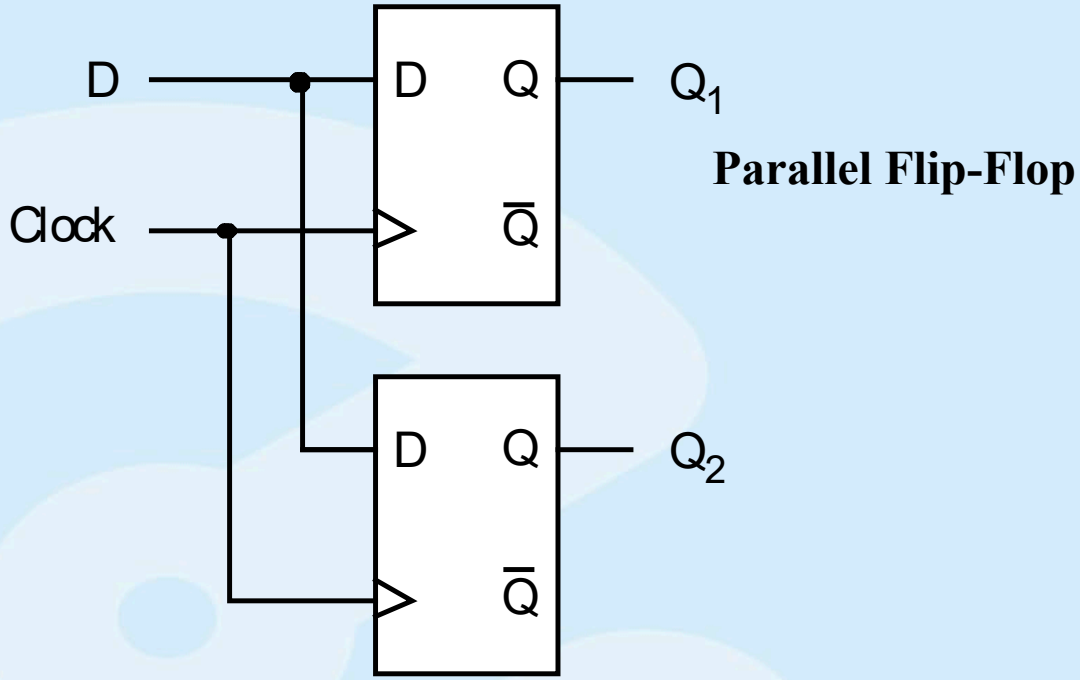
After posedge clk ⇒ foo = 2; bar = 2	After posedge clk ⇒ foo = 2; bar = 2

❖ *Từ ví dụ trên, chỉ sử dụng non-blocking statement trong **always**, còn blocking statement thì trong **always_comb** nhưng với sự thông hiểu datapath của mạch.*

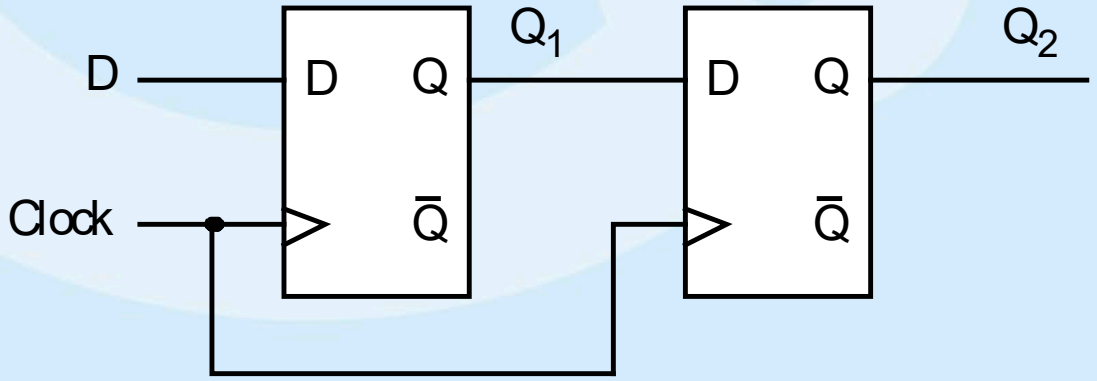
1. Always

```
module blocking_assign (
        input logic       D_i,
        input logic       clock_i,
        output logic      Q1_o,
        output logic      Q2_o
);
        always @(posedge Clock_i)
        begin
                Q1_o = D_i;
                Q2_o = Q1_o;
        end
endmodule

module blocking_assign (
        input logic       D_i,
        input logic       clock_i,
        output logic      Q1_o,
        output logic      Q2_o
);
        always @(posedge Clock_i)
        begin
                Q1_o <= D_i;
                Q2_o <= Q1_o;
        end
endmodule
```

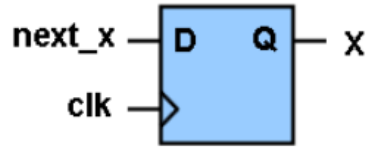


Parallel Flip-Flop

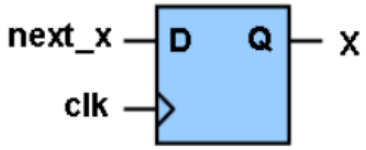


1. Always

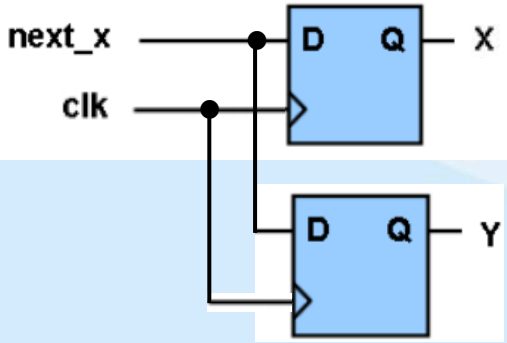
```
always @( posedge clk )
begin
   x = next_x;
end
```



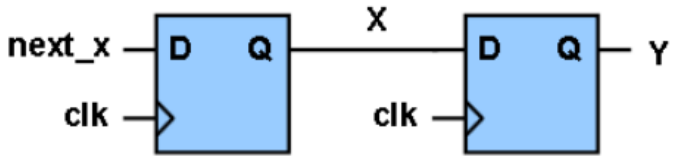

```
always @( posedge clk )
begin
   x <= next_x;
end
```




```
always @( posedge clk )
begin
   x = next_x;
   y = x;
end
```




```
always @( posedge clk )
begin
   x <= next_x;
   y <= x;
end
```



2. Always_ff

- Từ ví dụ trên, always có thể tạo ra D-FlipOop như sau:

```
always @(posedge clk) begin : proc_dff
  q <= d;
end
```

- Tuy nhiên, để báo cho compiler biết ở đó người thiết kế muốn tạo Flipflop, **always_ff** sẽ được sử dụng thay cho **always** thông thường.

```
always_ff @(posedge clk) begin : proc_dff
  q <= d;
end
```

❖ *Luôn luôn sử dụng **always_ff** khi thiết kế mạch có liên quan tới flipflop hoặc register.*

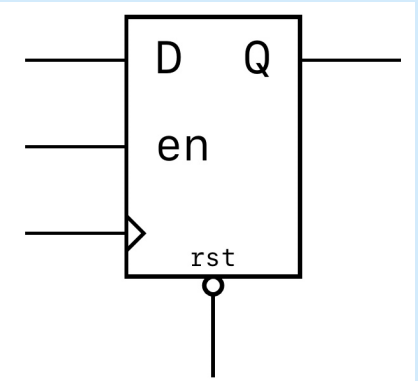
3. Sequential Logic

- Mạch sequential cơ bản là mạch combinational nhưng có phần tử memory (nhớ), điều này khiến ngõ ra (thông thường) sẽ thay đổi khi có sự thay đổi phần tử lái của memory (tín hiệu thay đổi (latch) hoặc xung clock). Ở đây chỉ xét xung clock.
- Như vậy, một mạch sequential khi viết bằng SystemVerilog sẽ có dạng như sau:

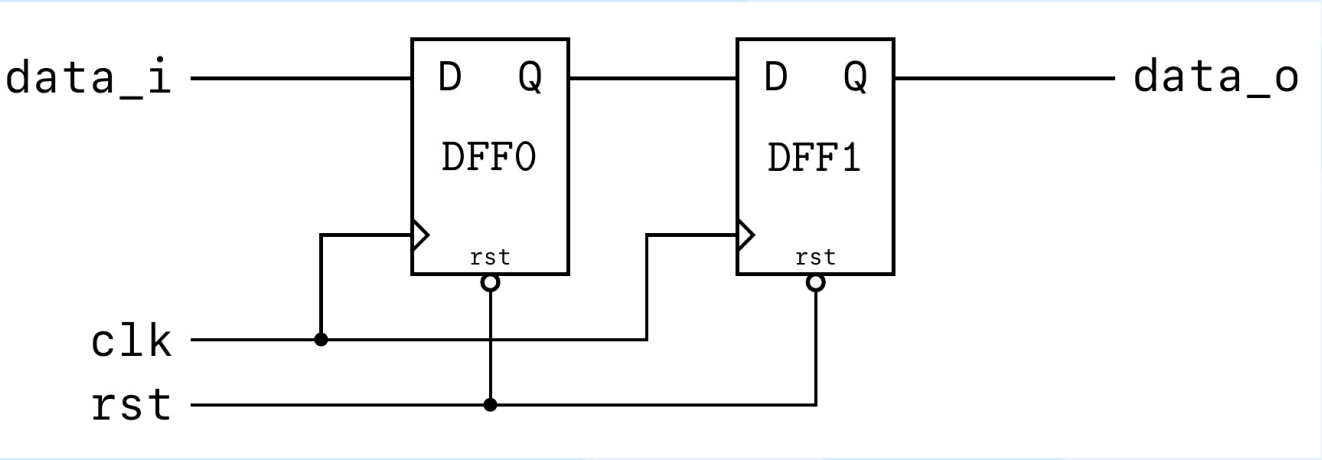
```
1   module sample (
2     input logic ...,
3   ...
4     output logic ...
5   );
6     // local declaration
7     logic ...
8
9     // input combinational logic
10    assign ...
11    always_comb begin : proc_input
12      ...
13    end
14
15    // DFF
16    always_ff @(posedge clk_i) begin : proc_dff
17      ...
18    end
19
20    // output combinational logic
21    assign ...
22    always_comb begin : proc_output
23      ...
24    end
25
26  endmodule : sample
```

4. Bài Tập

1. Thiết kế DFF có port synchronous reset và enable.



2. Thiết kế mạch synchronizer sử dụng 2 DFF có dạng như hình sau.



3. Thiết kế mạch counter 4-bit đếm lên nếu **en** = 1, ngõ ra **counter** cho biết giá trị hiện tại, ngõ ra này luôn bằng 0 nếu **rst** = 0.

4. Thiết kế mạch ALU 8-bit với ngõ ra ổn định bằng DFF.
(Sử dụng lại module đã thiết kế ở câu 4 phần 1)

II. Thiết Kế FSM – Máy Trạng Thái

1. Enum

- Nếu như logic chỉ có hai trạng thái 0, 1 được sử dụng (bỏ qua x, z) thì nếu muốn tạo nhiều trạng thái, ví dụ đèn giao thông có ba trạng thái RED, YELLOW, và GREEN. Ngoài việc khai báo 2 bit cho data, các giá trị RED, YELLOW, GREEN còn cần phải được đặt là localparam ứng với giá trị tự đặt.

```
localparam RED    = 2'b00;
localparam YELLOW = 2'b01;
localparam GREEN  = 2'b10;

logic [1:0] light;
```

- Tuy nhiên, SystemVerilog cung cấp một phương tiện giúp khai báo dễ hơn và dễ đọc hơn.

```
typedef enum logic [1:0] {
  RED    = 2'b00;
  YELLOW = 2'b01;
  GREEN  = 2'b10
} light_e;

light_e light;
```

```
typedef enum logic [1:0] {
  RED;
  YELLOW;
  GREEN
} light_e;

light_e light;
```

- **typedef** giúp tạo User-defined data type, **logic[1:0]** là storage type, do ở đây có 3 giá trị nên cần 2 bit.
- Lúc này, **light_e** là một data type, có 3 giá trị RED, YELLOW, và GREEN. Khai báo **light** có data type là **light_e**.
- Nếu không gán giá trị cho items trong lúc khai báo enum, khi synthesize, compiler sẽ tự tìm đường tối ưu hơn hết cho code.

1. Enum

- Như bài tập thiết kế ALU, ta có thể dùng **enum** cho tiện việc thảo code trong **always_comb**.

```
typedef enum logic [1:0] {
  NONE = 2'b00;
  ADD  = 2'b01;
  AND  = 2'b10;
  OR   = 2'b11;
} operation_e;

operation_e sel;

always_comb begin : proc_alu
  case (sel)
    NONE: ...
    ADD:  ...
...
```

2. FSM

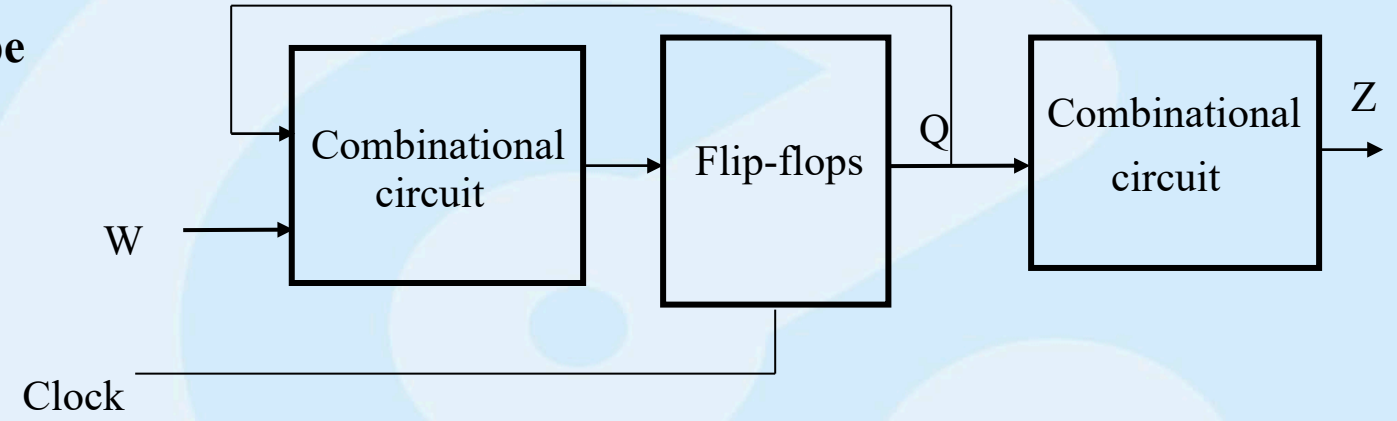
- FSM nhìn chung cũng chỉ như mạch sequential, bên dưới là một template cho việc code FSM.

```
1       module sample (
2         input logic ...,
3       ...
4         output logic ...
5       );
6         // local declaration
7         logic ...
8         typedef enum ...
9
10        // next-state combinational logic
11        assign ...
12        always_comb begin : proc_next_state
13          ...
14        end
15
16        // register
17        always_ff @(posedge clk_i) begin : proc_reg
18          ...
19        end
20
21        // output combinational logic
22        assign ...
23        always_comb begin : proc_output
24          ...
25        end
26
        endmodule : sample
```

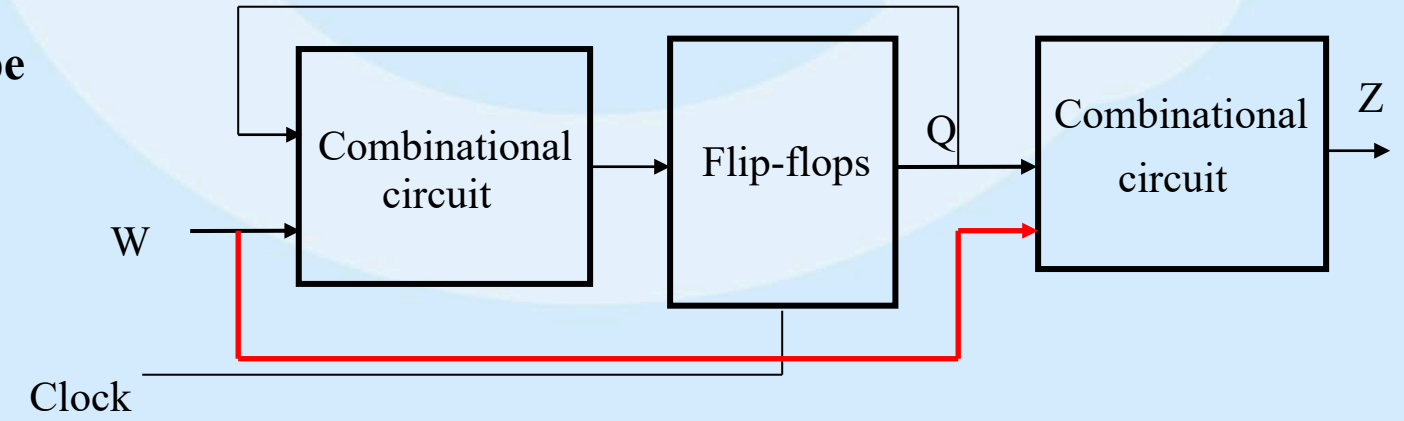
2. FSM

- General diagram of FSMs
- Two types of FSMs: Moore and Mealy
- Comparison between Moore and Mealy
- Introduction to Verilog for FSMs

Moore-type



Mealy-type

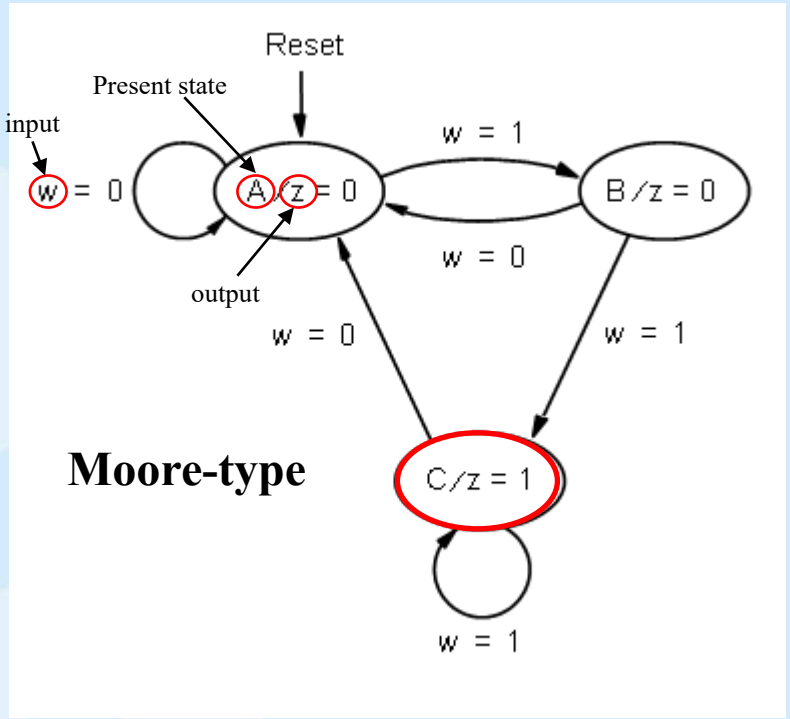


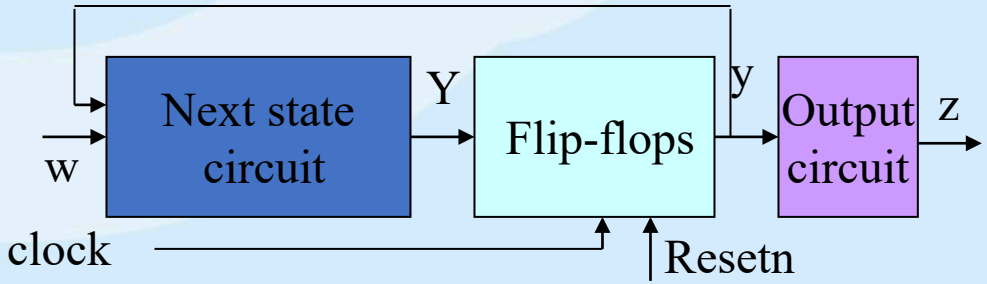
2. FSM

```
module simple (
    input logic Clock,
    input logic Resetn,
    input logic w,
    output logic z;
    );
    logic [2:1] y, Y;
    parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;
    // Define the next state combinational circuit
    always_comb  begin
        case (y)
            A: if (w)   Y = B;
               else     Y = A;
            B: if (w)   Y = C;
               else     Y = A;
            C: if (w)   Y = C;
               else     Y = A;
            default:    Y = 2'bxx;
        endcase
        end
    // Define the sequential block
    always_ff @(negedge Resetn or posedge Clock)
        if (Resetn == 0)            y <= A;
        else                        y <= Y;
    // Define output
    always_comb begin
        z = (y == C);
    end
Endmodule
```

y: present state
Y: next state

State assignment





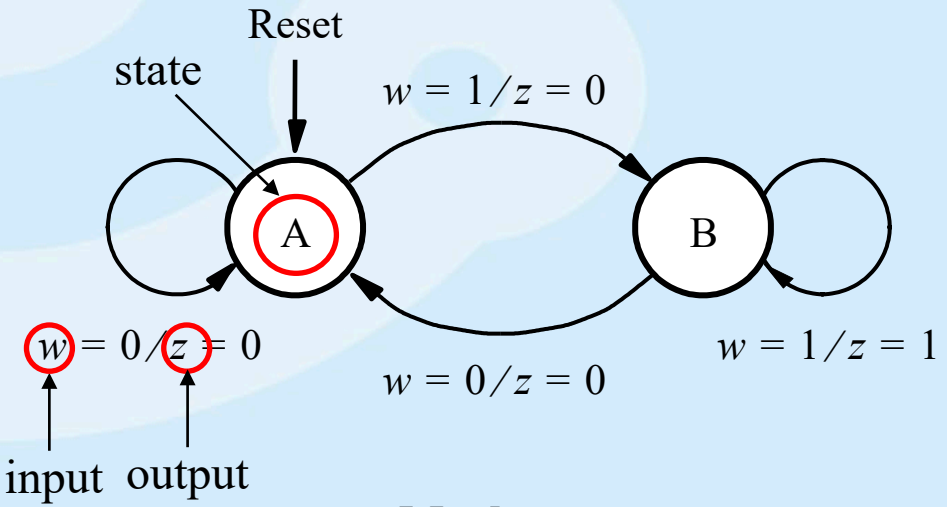
2. FSM

```
module mealy (
        input logic Clock,
        input logic Resetn,
        input logic w,
        output logic z
);
        logic y, Y;
        parameter A = 0, B = 1;
        // Define the next state and output
        //combinational circuits
        always_comb  begin
            case (y)
                A:    if (w)
                            begin
                            z = 0;  Y = B;
                            end
                      else  // w = 0
                            begin
                            z = 0;  Y = A;
                            end
                B:    if (w)
                            begin
                            z = 1;  Y = B;
                            end
                      else
                            begin
                            z = 0;  Y = A;
                            end
            endcase
        end
```

2 parallel blocks:

1) **Always_comb** block:
   combinational circuit for *next state* and *output*
2) **Always_ff** block:
   update states

```
// Define the sequential block
        always_ff @(negedge Resetn or posedge Clock)
            if (Resetn == 0)    y <= A;
            else                y <= Y;
endmodule
```



Mealy-type

3. Bài Tập

1. Thiết kế FSM có giản đồ trạng thái như sau.