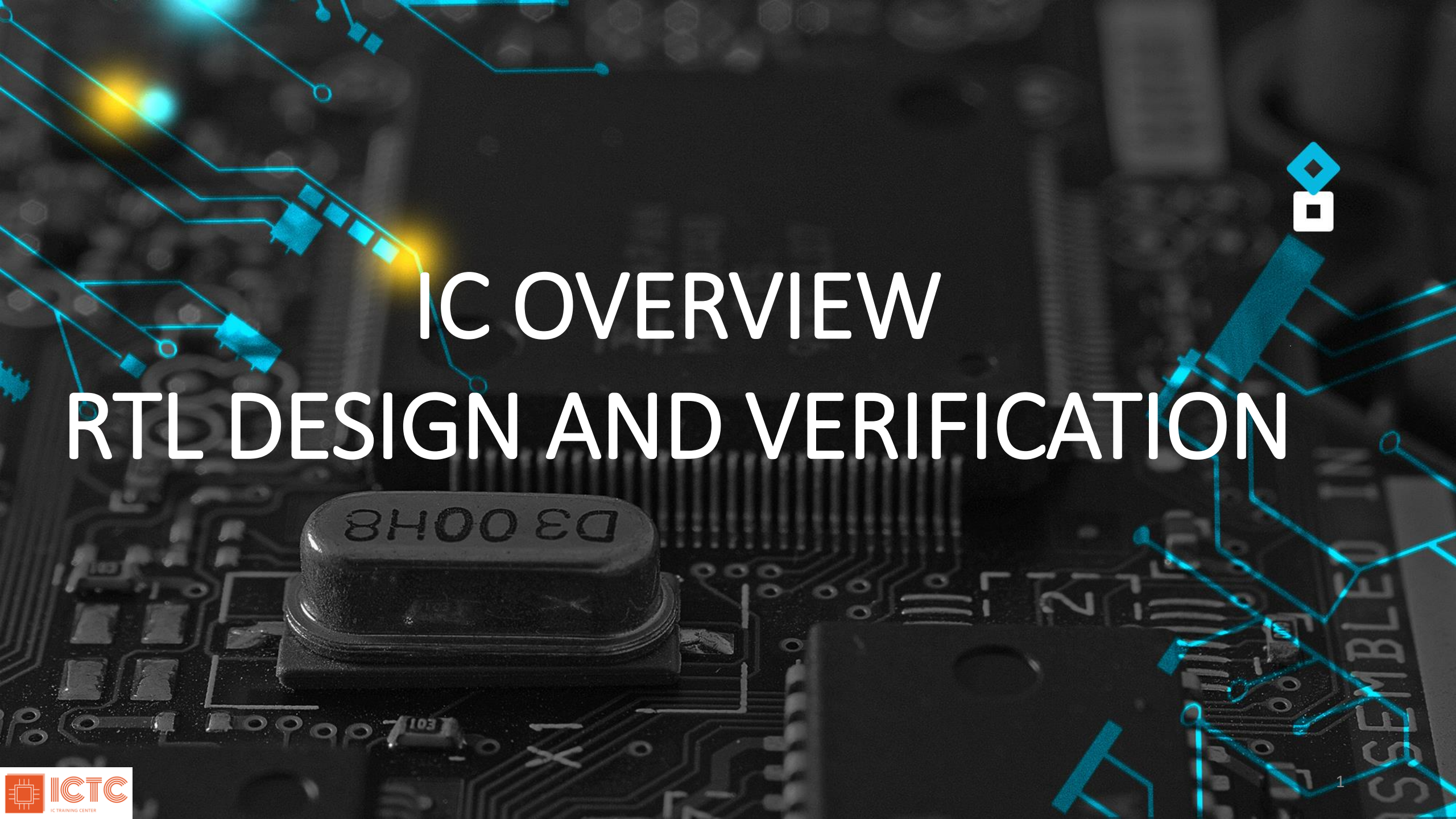
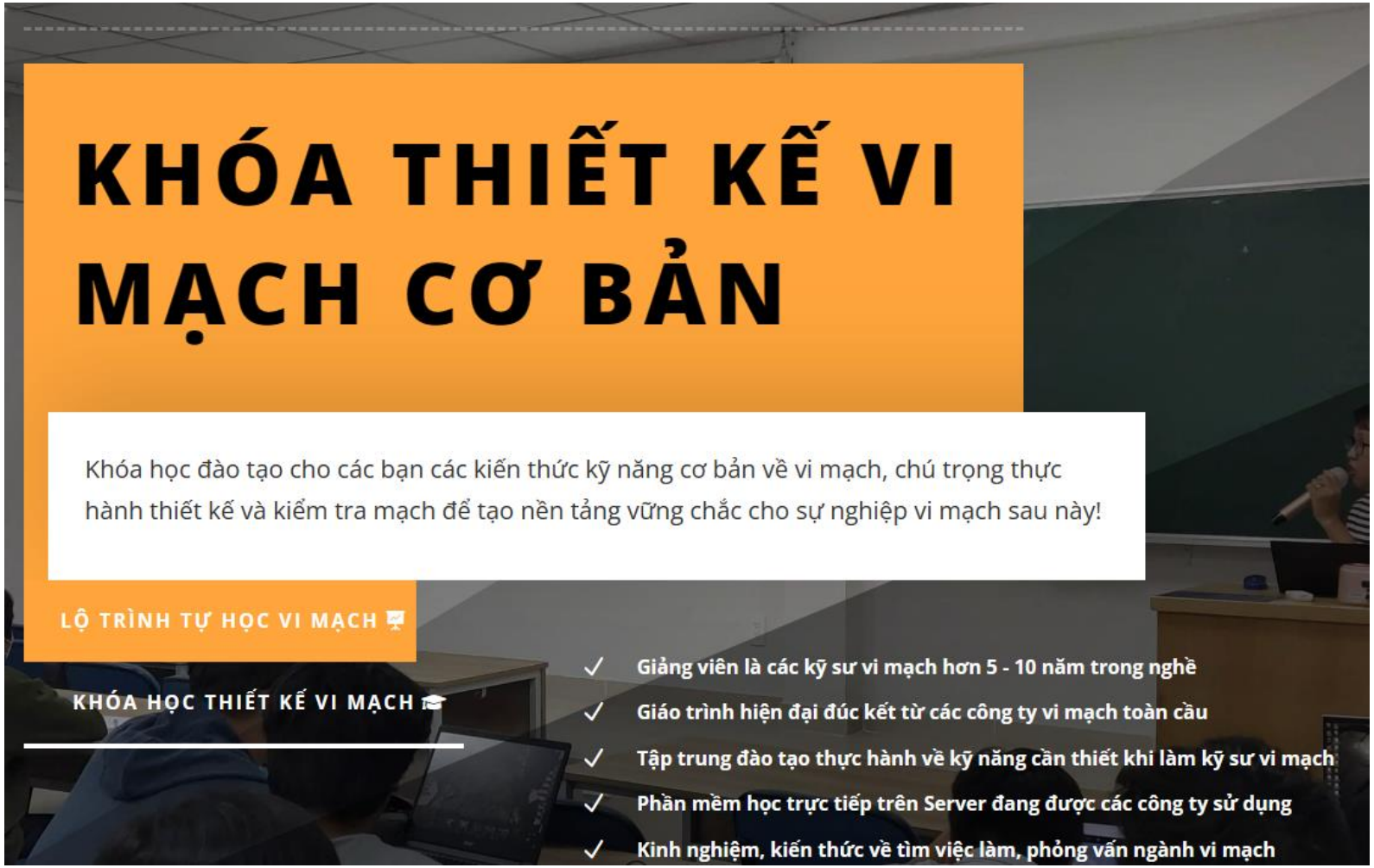


IC OVERVIEW
RTL DESIGN AND VERIFICATION

COURSE INTRODUCTION

[Khóa Học Thiết Kế Vi Mạch Cơ Bản - Trung Tâm Đào Tạo Thiết Kế Vi Mạch ICTC]



COURSE INTRODUCTION

SUMMARY

HOMEWORK

QUESTION

SELF-LEARNING

Session 8: Verilog Fundamental – Part 3 Always block

1. Block statement
2. Combinational logic
3. Procedural assignment
4. Reg type
5. Always block
6. Mux design
7. Encoder/decoder

INTRODUCE OF BLOCK STATEMENT

- Block statement is a way to group multiple statements together into a single logical block.
- This is useful for organizing code and controlling the scope of variables. Below is an example using begin-end block statement.

```
module example;
  // Variable declaration outside the block
  integer a;

  initial begin
    // Variable declaration inside the block
    integer b;
    // Statements inside the block
    a = 1;
    b = 2;
    $display("a = %0d, b = %0d", a, b);
  end
endmodule
```

REG DATA TYPE

- Reg data type is used to represent:
  - storage elements: flip-flops, latches. (will be described in session 9)
  - combinational logic in procedures. (described in today session)
- It can hold value during assignments and keep value until next assignment.
- Reg data type can hold value does not means it is “memory”. If it represents for combinational logic, its value is memorized by the simulator only.
- Keyword: reg
  - Default: one-bit value
  - Default value: x (unknown).
  - Examples :

```
reg enable;
reg [15:0] data;      //vector
```

PROCEDURAL BLOCKING ASSIGNMENT

- This kind of assignment is inside a procedural block (always, initial)
- Blocking: the current assignment statement blocks the other assignments
- **Syntax**: y = expression;
- **Example**: (1) b = a + 1; //This statement will block the next statement
  (2) c = b + 2;

  Assume that a = 1, b = 1 → (1) is executed first, then b = 2, (2) is executed after, then c = 4
- **Non-blocking**: all assignment run in parallel (will be described in session 9)
- The LHS of this assignment need to be "reg" type.

Always make sure that bit-width of LHS and RHS are equal to avoid unexpected issues

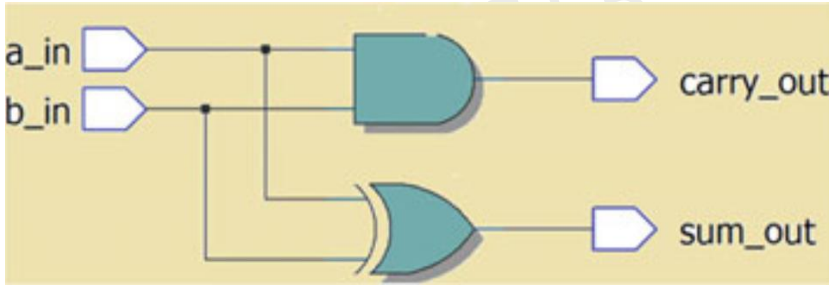
COMBINATIONAL LOGIC DESIGN

- Combinational logic is implemented by using the logic gates and in the combinational logic, output is the function of present input.
- The goal of a designer is always to implement the logic using minimum number of logic gates or logic cells, minimization techniques are K-map, Boolean.
- Combinational logics are created by continuous assignment (using assign) or procedural assignment using “blocking assignment” (=).

COMBINATIONAL LOGIC DESIGN

We already knew how to design combinational logic using continuous assignment in previous sessions.

```
module half_adder(a_in, b_in, sum_out, carry_out);
   input  a_in;
   input  b_in;
   output sum_out;
   output carry_out;

   wire a_in, b_in, sum_out, carry_out;

   assign sum_out = a_in ^ b_in;
   assign carry_out = a_in & b_in;

endmodule
```

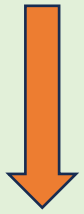


Synthesized half adder

COMBINATIONAL LOGIC DESIGN

- Combinational logic can also be generated by procedural assignment with blocking assignment "=" inside a procedure using always block.
- The assignment in the procedural assignment (for combinational logic) is executed in-order, not parallel.

```
always @( sesivity list) begin
   [statement1]
   [statement2]
   [statement3]
end
```



Statements are executed in-order using blocking assignment "="

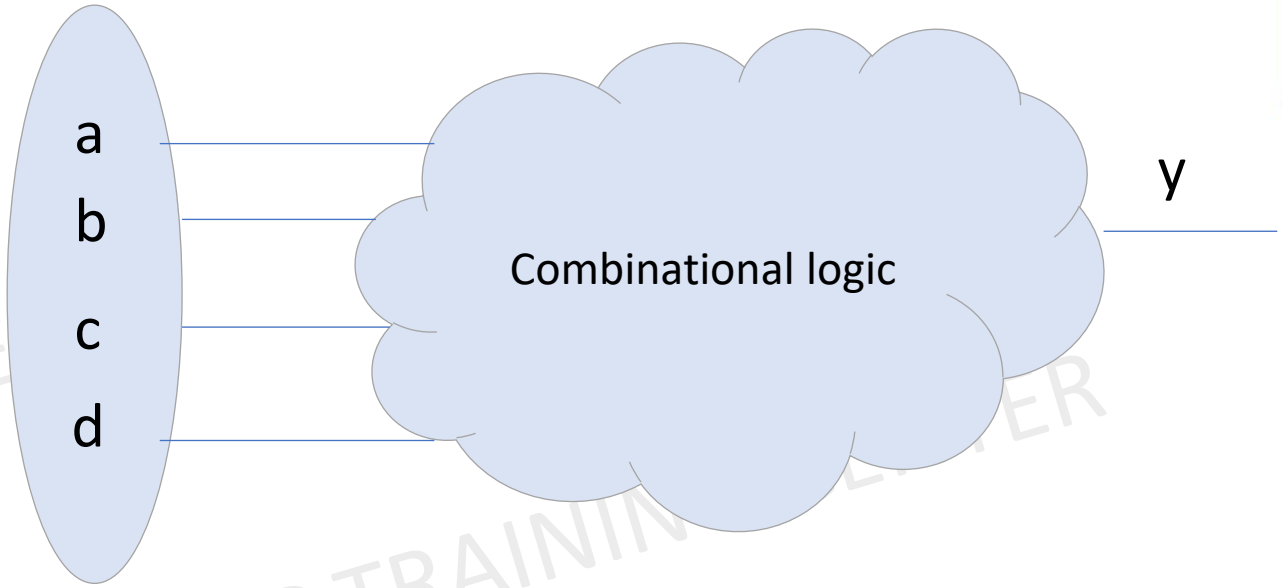
PROCEDURAL ALWAYS BLOCK

- The always blocks are enabled at the beginning of a simulation at time 0. They shall execute repeatedly. They could be end when the simulation is terminated.
- An always block is one of procedural block, statement inside an always block are executed sequentially
- @ is sensitivity list: the code inside always block is executed only when any signal in the sensivity list is changed.
- the output of signal inside always block should be reg.

```
module abc(a,b);

.....

//Combination logic
always @ (sensitivity list)
   begin
     [statement1];
     [statement2];
     [statement3];
   end
end

endmodule
```

PROCEDURAL ALWAYS BLOCK

```
//Combination logic
reg y;
always @ (a or b or c or d)
   begin
      y = …. ;   //blocking assignment
   end
end
```



Whenever a, b, c or d, which is input to the logic is updated, the logic in always construct is executed because these signals are written in the sensitivity list.

PROCEDURAL ALWAYS BLOCK

Restriction

Some rules when writing always construct to generate combinational logic:

- One always construct can define many output variables. However, define many variables in one big always construct is not recommended.
- Do not use non-blocking assignment (<=) for combinational logic.
- Must assign values to all the variables in every branches. Otherwise, it will create latch.
- Do not assign one variable in different always constructs.

```
always @ (a)
begin
   if( a == 1'b1) begin
      y = 2'b10;
   end
            <------- else is missing
end
```

```
always @ (a)
begin
   if( a == 1'b1) begin
      y = 2'b10;
   end
end

always @ (b)
begin
   if( b == 1'b1) begin
      y = 2'b01;
   end
end
```

PROCEDURAL ALWAYS BLOCK

Practice

Practice: design half-adder using always block instead of continuous assignment.

Use previous testbench to check pass/fail.

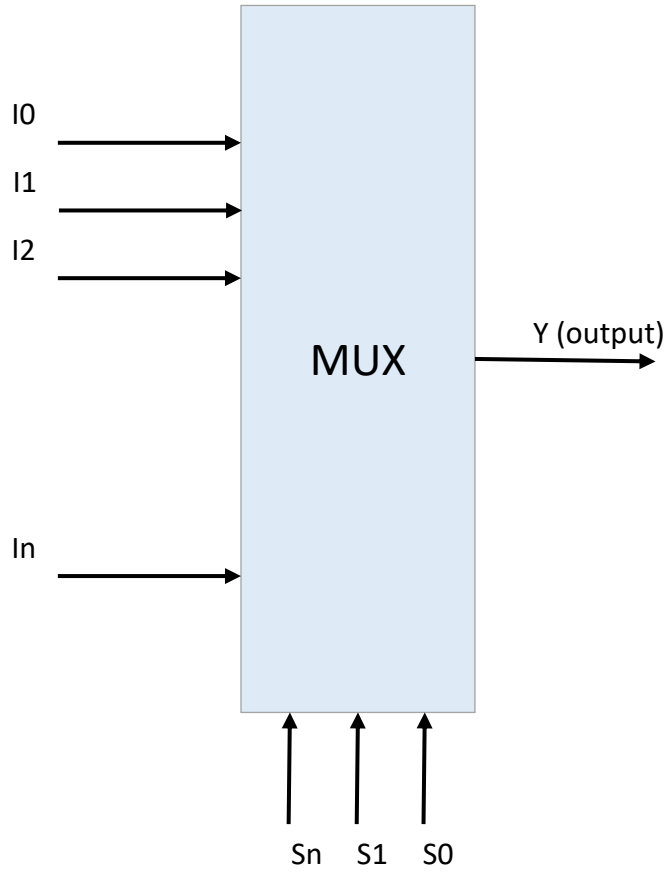
MULTIPLEXER (MUX)

- Multiplexers are used to select one of the input from many.
- Multiplexers are also called as universal logic and terminology used in the practical world is MUX
- The formula to express the relations between bit-width of selector and number of input:

  number of inputs = 2 ^ (selector's bit-width)
- Example:

  4-input mux needs 2-bit select signal

  8-input mux needs 3-bit select signal

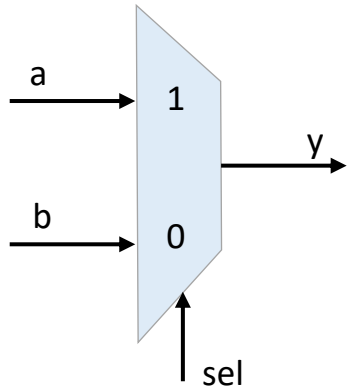


DESIGN MUX

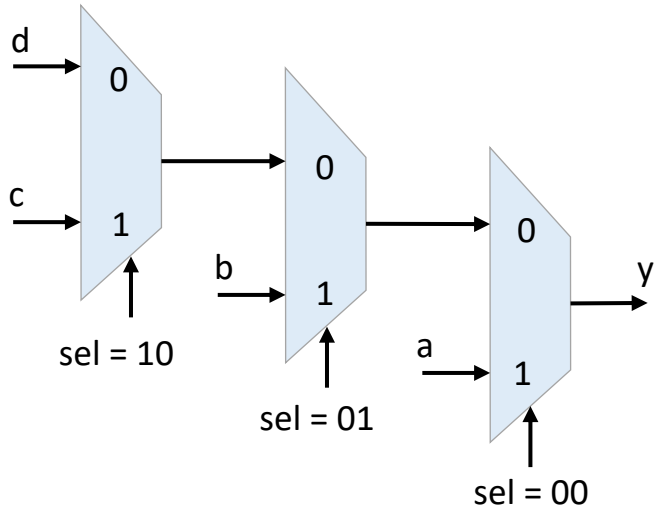
BY CONTINUOUS ASSIGNMENT

- Multiplexer can be designed by continuous assignment using conditional operator

```
wire y;

assign y = sel ? a : b;
```




```
wire y;
wire [1:0] sel;

assign y = (sel == 2'b00) ? a :
           (sel == 2'b01) ? b :
           (sel == 2'b10) ? c : d;
```



DESIGN MUX

BY PROCEDURAL ASSIGNMENT USING IF STATEMENT

- Multiplexer can be designed by procedural assignment (blocking) using if-else statement.
- If-else statements need to be inside procedure.

```
always @ (sensivity list) begin
   if( expression )
      [statement]
   else if( expression )
      [statement]
   …more else if blocks
   else
      [statement]
end
```

Begin/end is optional when there is only 1 statement under if-else.

Begin/end is mandatory for multiple statements under if-else

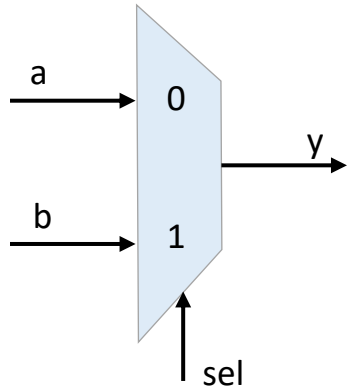
```
always @ (sensivity list) begin
   if( expression ) begin
      [statement]
      [statement]
   end
   else if( expression ) begin
      [statement]
      [statement]
   end
   …more else if blocks
   else begin
      [statement]
    end
end
```

DESIGN MUX

BY PROCEDURAL ASSIGNMENT USING IF STATEMENT

Example of mux 2 to 1:

```
wire a, b, sel;
reg y;

always @ (a or b or sel) begin
   if(sel == 0)
      y = a;
   else
      y = b;
end
```



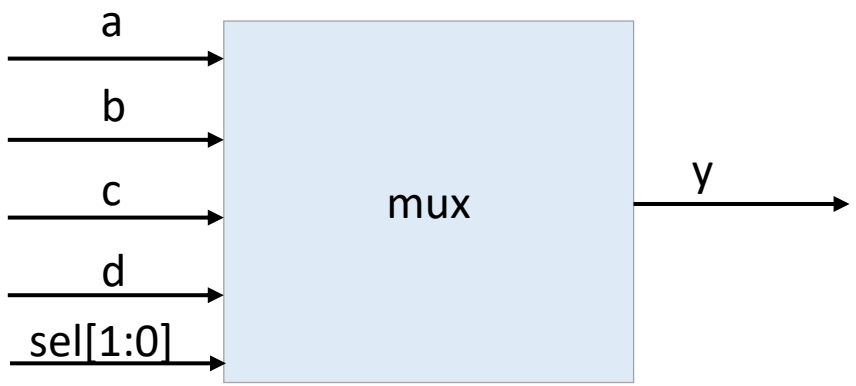
Need begin-end if there are more than 1 statement under if-else

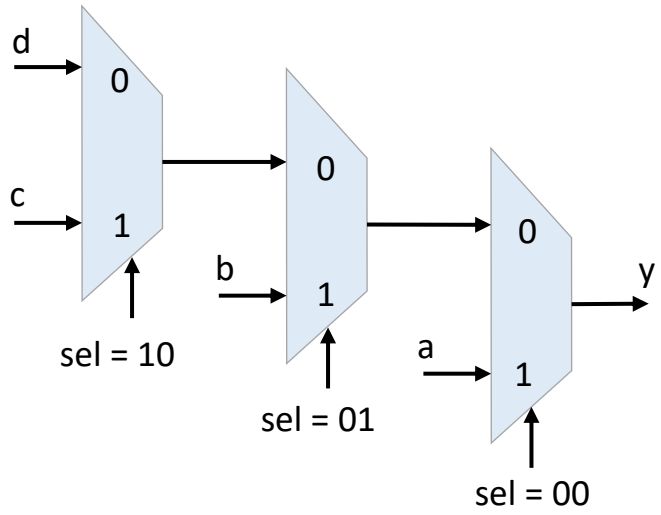
DESIGN MUX

BY PROCEDURAL ASSIGNMENT USING IF STATEMENT

<u>Practice</u>: Design below mux using if-else statement

1. Make 08_ss8 folder under your home directory
2. Copy /ictc/student-data/share/teacher/08_ss8/mux/ to your folder
3. Create the mux.v under rtl folder and writing your code. The port name need to be same as the diagram.
4. Go to sim and compile, run to check the result.





DESIGN MUX

BY PROCEDURAL ASSIGNMENT USING CASE STATEMENT

Mux can be designed by case statement

```
always @ (sensivity list) begin
   case( expression )
      value_0:
         [statement]
      value_1:
         [statement]
       … more value …
      default:
         [statement]
   endcase
end
```

Begin/end is optional when there is only 1 statement under if-else.

Begin/end is mandatory for multiple statements under if-else

```
always @ (sensivity list) begin
   case( expression )
      value_0: begin
         [statement]
         [statement]
      end
      value_1: begin
         [statement]
         [statement]
      end
       … more value …
      default: begin
         [statement]
         [statement]
      end
   endcase
end
```

DESIGN MUX

BY PROCEDURAL ASSIGNMENT USING CASE STATEMENT

Example: mux2to1 using case

```
//1st method
wire a,b,sel:
reg  y;

always @ (a or b or sel) begin
   case( sel )
      1'b0:     y = a;
      default: y = b; //sel=1'b1
   endcase
end
```

```
//2nd method
wire a,b,sel:
reg  y;

always @ (a or b or sel) begin
   case( sel )
      1'b0:     y = a;
      1'b1:     y = b;
      default: y = <any value>;
   endcase
end
```

```
//3rd method
wire a,b,sel:
reg  y;

always @ (a or b or sel) begin
   case( sel )
      1'b0:     y = a;
      1'b1:     y = b;
   endcase
end
```

All cases will give the same result during synthesis because they are “full case”.
However, using 1st and 2nd method are recommended and is a good practice (using default).
Also, some RTL DRC tools will give error/warning on not using “default” in case statement.

DESIGN MUX

BY PROCEDURAL ASSIGNMENT USING CASE STATEMENT

Analyze more on 1st and 2nd method (full-case with default)

```
//1st method
wire a,b,sel:
reg  y;

always @ (a or b or sel) begin
   case( sel )
      1'b0:     y = a;
      default: y = b; //sel=1'b1
   endcase
end
```

- When sel is x or z (unexpected), y is assigned to b.
- If "b" is the expected result, we may not see the issue of sel. The issue only occurs when expected result is "a".
- This kind of coding style is similar to the if-else mux using continuous assignment and conditional operator.

```
//2nd method
wire a,b,sel:
reg  y;

always @ (a or b or sel) begin
   case( sel )
      1'b0:     y = a;
      1'b1:     y = b;
      default: y = 1'bx;
   endcase
end
```

- In default case, y can be assigned to any value, 0 or 1 or 1'bx.
- Assign 0 or 1 to the y in the default case can make the simulation becomes fail when sel is x or z.
- Assign 1'bx to the result will make x-propagation to the later logic in simulation. It can help to trace x and file the root cause easily by EDA tools.
- The code coverage will be lower and need to be waived because we can not cover the default case

Both cases will have the same result after synthesis.

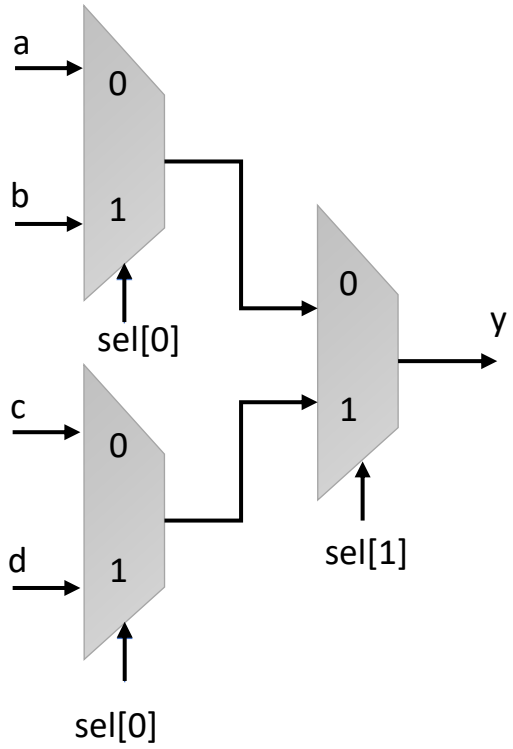
Which one is chosen depends on the coding rule of each company.

DESIGN MUX

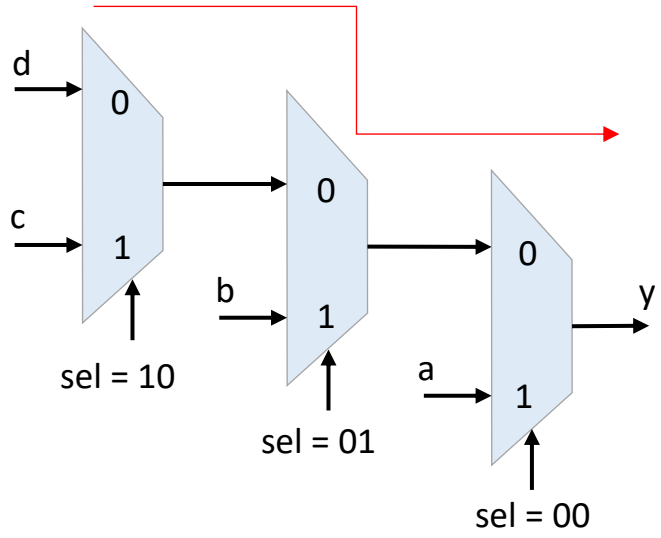
BY PROCEDURAL ASSIGNMENT USING CASE STATEMENT

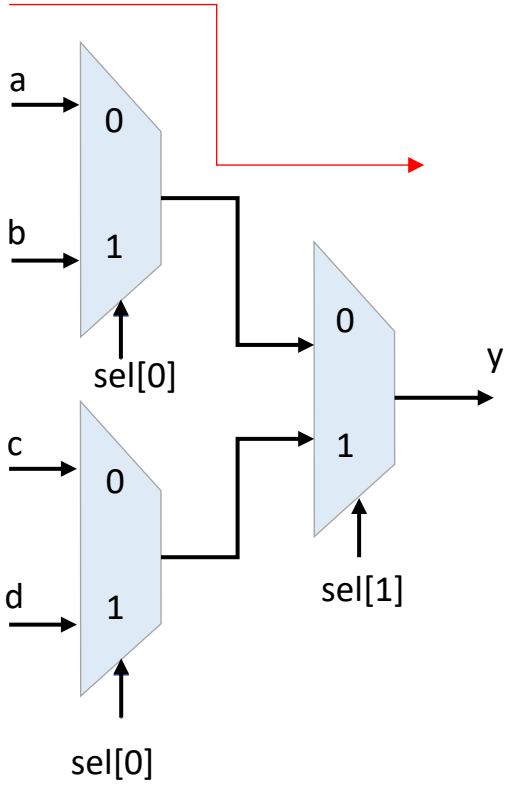
Example: mux4to1 using case

```
always @ (a or b or c or d or sel) begin
   case( sel )
      2'b00:    y = a;
      2'b01:    y = b;
      2'b10:    y = c;
      default:  y = d;  //2'b11
   endcase
end
```



IF-ELSE VS CASE MUX





The most critical path of if-else mux diagram passes through 3 muxes.

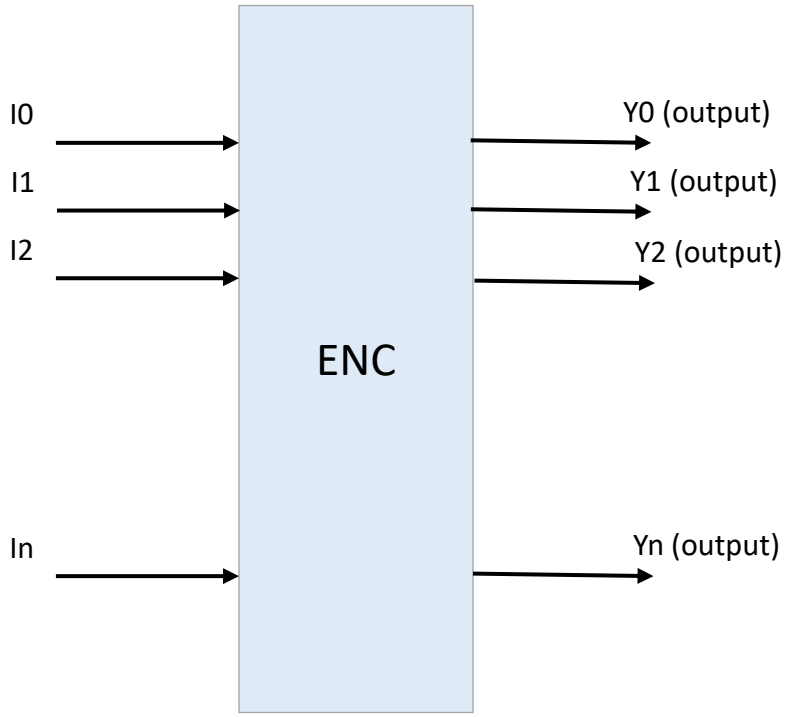
The most critical path of case mux diagram passes through 2 muxes.

→ Case statement is faster

→ If-else statement is slower, but can be used for prioritizing the conditions.

ENCODER

An encoder is a combinational logic circuit that converts information from 2^n input lines to an n-bit binary code.

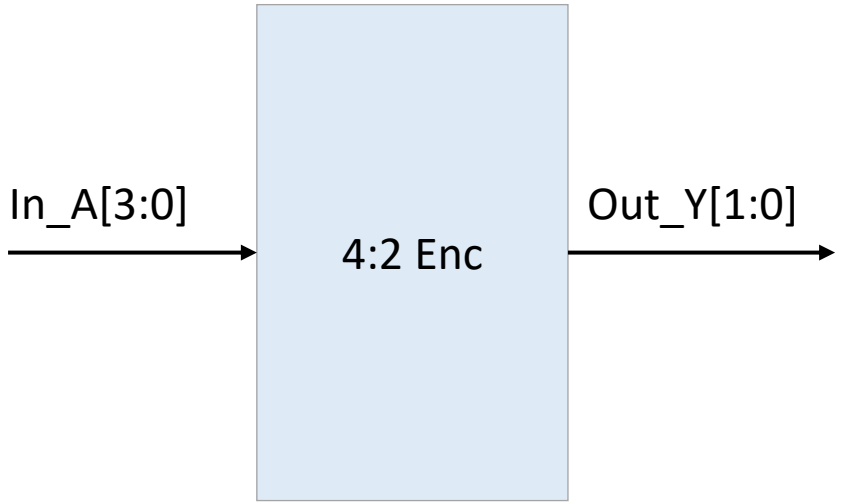


ENCODER

EXAMPLE ENCODER 4:2

Example: 4:2 encoder

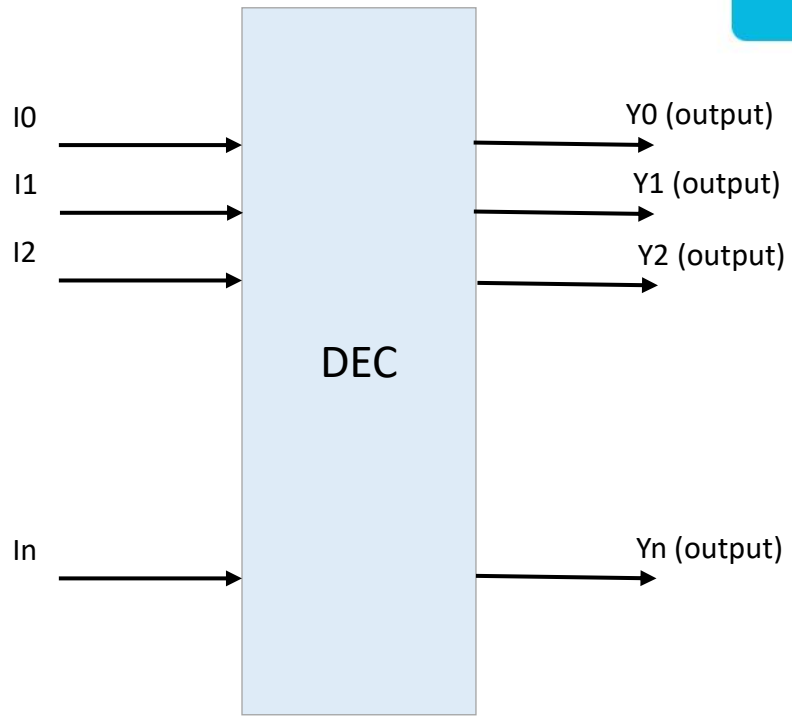
In_A[3]	In_A[2]	In_A[1]	In_A[0]	Out_Y[1]	Out_Y[0]
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1




```
always @ (In_A) begin
   case(In_A)
      4'b0001 : Out_Y = 2'b00;
      4'b0010 : Out_Y = 2'b01;
      4'b0100 : Out_Y = 2'b10;
      4'b1000 : Out_Y = 2'b11;
      default  : Out_Y = 2'b00;
   endcase
end
```

DECODER

- A decoder is a combinational logic circuit that converts binary information from 'n' input lines to a maximum of 2^n unique output lines.
- Decoders are often used in applications where you need to select one of many outputs based on a binary input value, such as memory address decoding, data routing, and other digital signal processing tasks

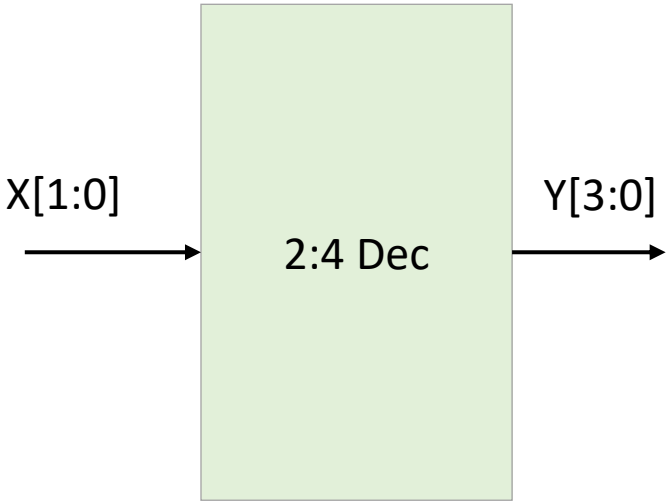


DECODER

EXAMPLE DECODER 2:4

Truth table for 2:4 Decoder

X2	X1	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

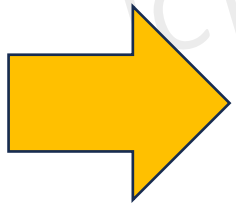



```
always @ (X) begin
   case(X)
      2'b00   : Y = 4'b0001;
      2'b01   : Y = 4'b0010;
      2'b10   : Y = 4'b0100;
      default : Y = 4'b1000;
   endcase
end
```

TIP

- Can use always @* instead of explicity listing a sensitivity list.
- The always @* construct automatically infers the sensitivity list by including all signals that appear in the right-hand side of assignments within the always block.
- This helps avoid mistakes related to incomplete sensitivity lists and makes the code easier to read and maintain.

```
//Combination logic
reg y;
always @ (a or b or c or d)
   begin
      y = …. ;   //blocking assignment
   end
end
```



```
//Combination logic
reg y;
always @*
   begin
      y = …. ;   //blocking assignment
   end
end
```

SESSION 8
SUMMARY

SUMMARY:

- Combinational logic can be described by either continuous assignment or procedural assignment.
- In Procedural assignment, the LHS need to be reg data type.
- Always block is a procedural block. Sensivity list is a list of signals. The code inside always block is executed whenever any signal in sensivity list is changed.
- Mux can be designed by conditional assignment, if-else or case statement.
- Encoder is combinational logic block that used to convert 2^n input to n output.
- Decoder is combinational logic block that used to convert n input to 2^n output.

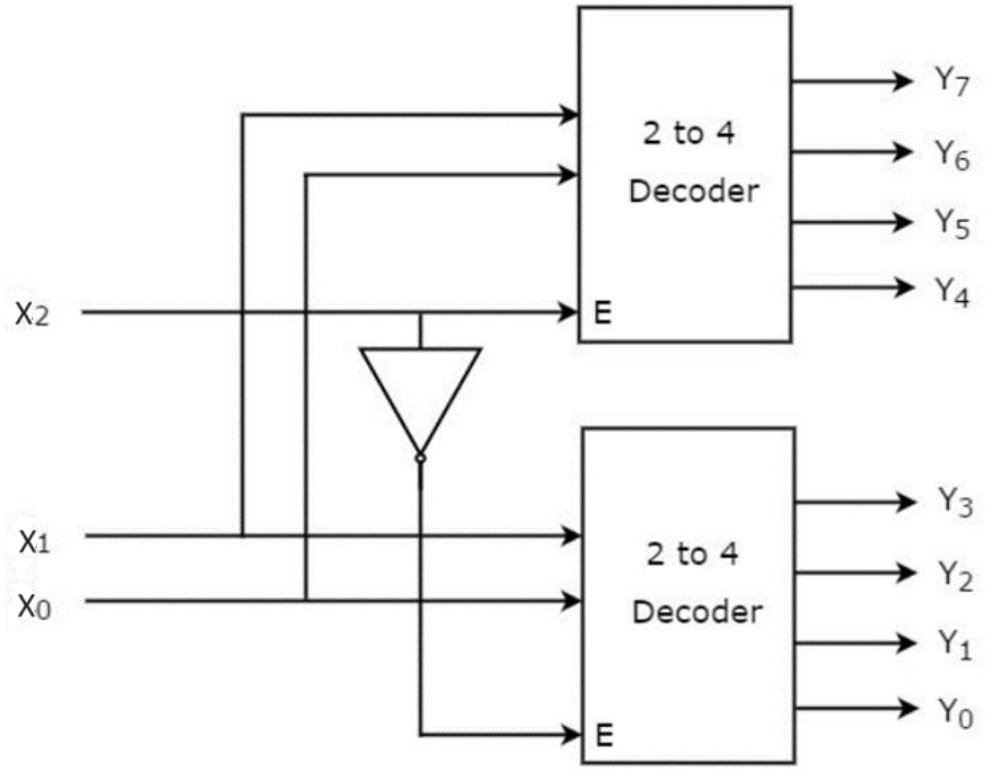
HOMEWORK

Homework1: Design 3:8 decoder using 2:4 decoder

A 3:8 decoder can be generated based on two 2:4 decoders as the right figure.

An additional E (enable) port is used for 2:4 decoder with below specification.

- E = 0: decoder is disabled. Output of decoder are 0.
- E = 1: decoder is enabled. It can output decoded value.

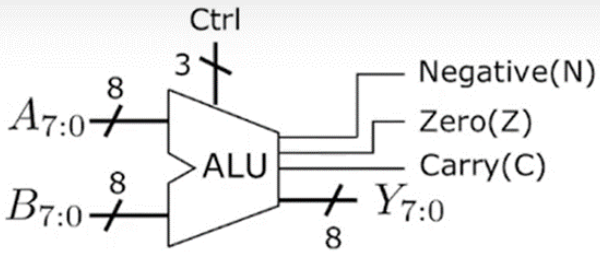


1. Copy /ictc/student-data/share/teacher/08_ss8/decoder_3to8/ to your folder
2. Create decoder_2to4.v and decoder_3to8.v under rtl folder and writing your code. The port name need to be same as the diagram.
3. Go to sim and compile, fix all the real errors/warnings, then run to check the result.

Session 8

Homework2(*): Design and verify for an ALU with below specification

1. Copy /ictc/student_data/share/teacher/08_ss8/alu to your 08_ss8 folder
2. Design alu.v follows below specification
3. Run simulation and check the result.



Ctrl	Y	Notes
000	A & B	Bitwise AND
001	A | B	Bitwise OR
010	A + B	Addition
110	A - B	Subtraction

- Carry Flag (C): set when there is a carry from the MSB of the result
- Negative Flag (N): set when a math operation results in a negative result
- Zero Flag (S): set with a math operation results in a zero result