

Lecture 1: Logic Design Basics, ModelSim

Xuan-Truong Nguyen

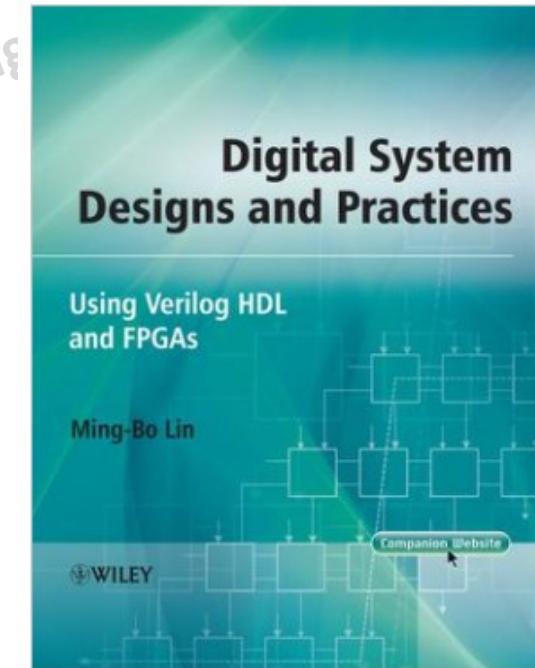


Before the class, Acknowledgment

- Digital Systems Design and Experiments (<https://ocw.snu.ac.kr/node/2390>)
- Digital System Designs and Practices: Using Verilog HDL and FPGAs @Wiley 2008
- Some slides are reused from the Digital Systems Design and Experiments

Digital Systems Design and Experiments

Lecture Notes	Calendar	Assignments	Exam	Study Materials
c_lecno	Title			files
1-1	Introduction			6625.pdf
1-2	Introduction / Digital Design Methodology			6626.pdf
2	Structural Modeling			6627.pdf
3	Dataflow Modeling			6628.pdf
4	Behavioral Modeling			6629.pdf
5	Tasks, Functions, and UDPs			6630.pdf
8	Combinational Logic Modules			6631.pdf
9	Sequential Logic Modules			6632.pdf
10	Design Options of Digital Systems			6633.pdf
11	System Design Methodology			6634.pdf
12	Synthesis			6635.pdf
13	Verification			6636.pdf
15-1	Design Examples			6637.pdf
15-2	Design Examples / CPU Design Example			6639.pdf
16	Design for Testability			6638.pdf



Lecture plan

- Today we will:
 - Review the numerical data types, and hardware description language (HDL)
 - Describe the HDL-based design flow
 - Describe the basic features of the modules in Verilog HDL
 - Describe how to model a design in Verilog HDL
 - Describe how to simulate/verify a design using Verilog HDL
- Labs
 - Lab 1: Hello ModelSim
 - Lab 2: Encoder
 - Lab 3: Counter

Numeric Data Types

- How is numeric data represented in modern computing systems?
- Data types
 - Integer
 - Fixed-point number
 - Floating-point number

Integer: Binary format

- Unsigned number

- n -bit: Range: $[0, 2^n - 1]$

- Signed number

- *Sign-Magnitude Representation*

- n -bit Range: $[-2^{n-1} - 1, 2^{n-1} - 1]$

- Both 00...0 and 10...0 represent 0

↙ ↘ ↙ ↘ ↗ ↖ ↗ ↖

- Two's Complement Representation

- n -bit Range: $[-2^{n-1}, 2^{n-1} - 1]$

- 00...0 represents 0

- 10...0 represents -2^{n-1}

0	0	1	1	0	0	0	1
x	x	x	x	x	x	x	x

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 49$$

Sign Bit

1	0	1	1	0	0	0	1
x	x	x	x	x	x	x	x

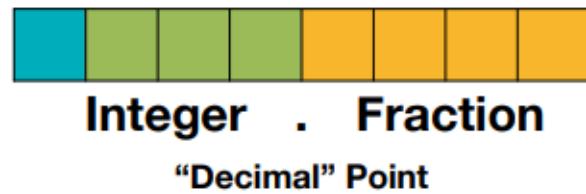
$$-2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -49$$

1	1	0	0	1	1	1	1
x	x	x	x	x	x	x	x

$$-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -49$$

Fixed-point number

- Similar to an integer format, a fixed-point number is decomposed in [sign, ibit, fbit]



This diagram shows the same 9-bit fixed-point number representation as above, but with binary values assigned to each bit position. The sign bit is 0 (blue). The integer bits are 0, 0, 1, 1 (green). The fraction bits are 0, 0, 0, 1 (orange). Below each bit is a multiplier: \times , \times . To the right of the fraction bits, the equation is shown: $-2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 3.0625$.

This diagram shows the same 9-bit fixed-point number representation as above, but with different binary values assigned to each bit position. The sign bit is 0 (blue). The integer bits are 0, 0, 1, 1 (green). The fraction bits are 0, 0, 0, 1 (orange). Below each bit is a multiplier: \times , \times . To the right of the fraction bits, the equation is shown: $(-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0) \times 2^{-4} = 49 \times 0.0625 = 3.0625$.

(using 2's complement representation)

Floating-Point Number

- Example: 32-bit floating-point number in IEEE 754

→ defined by association



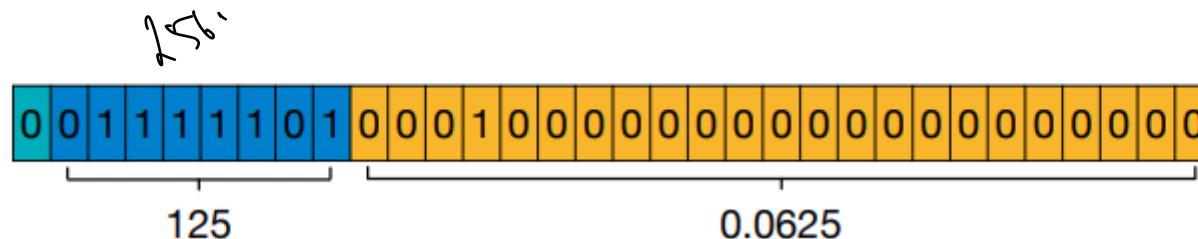
Sign 8 bit Exponent

23 bit Fraction

$$(-1)^{\text{sign}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent}-127} \quad \leftarrow \text{Exponent Bias} = 127 = 2^{8-1}-1$$

(significant / mantissa)

$$0.265625 = 1.0625 \times 2^{-2} = (1 + 0.0625) \times 2^{125-127}$$



Copyright

Floating-Point Number

- Exponent Width → Range; Fraction Width → Precision

IEEE 754 Single Precision 32-bit Float (IEEE FP32)	Exponent (bits)	Fraction (bits)	Total (bits)
	8	23	32
IEEE Half Precision 16-bit Float (IEEE FP16)	5	10	16
Brain Float (BF16)	8	7	16
Nvidia TensorFloat (TF32)	8	10	19
AMD 24-bit Float (AMD FP24)	7	16	24

more larger range of numbers compared to IEEE FP32

google ai

Numeric Data Types

- Question: What is the following IEEE half-precision (IEEE FP16) number in decimal?



- Sign: -
- Exponent: $10001_2 - 15_{10} = 17_{10} - 15_{10} = 2_{10}$
- Fraction: $1100000000_2 = 0.75_{10}$
- Decimal Answer = $-(1 + 0.75) \times 2^2 = -1.75 \times 2^2 = -7.0_{10}$

Copyright 2022. (차세대반도체 혁신공유대학 사업단)

Numeric Data Types

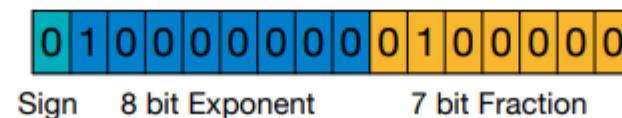
- Question: What is the decimal 2.5 in Brain Float (BF16)?

$$2.5_{10} = \underline{1.25}_{10} \times 2^{\underline{1}}$$

Exponent Bias = 127_{10}

↙ ↗ ↘

- Sign: +
- Exponent Binary: $1_{10} + 127_{10} = 128_{10} = 10000000_2$
- Fraction Binary: $0.25_{10} = 0100000_2$
- Binary Answer



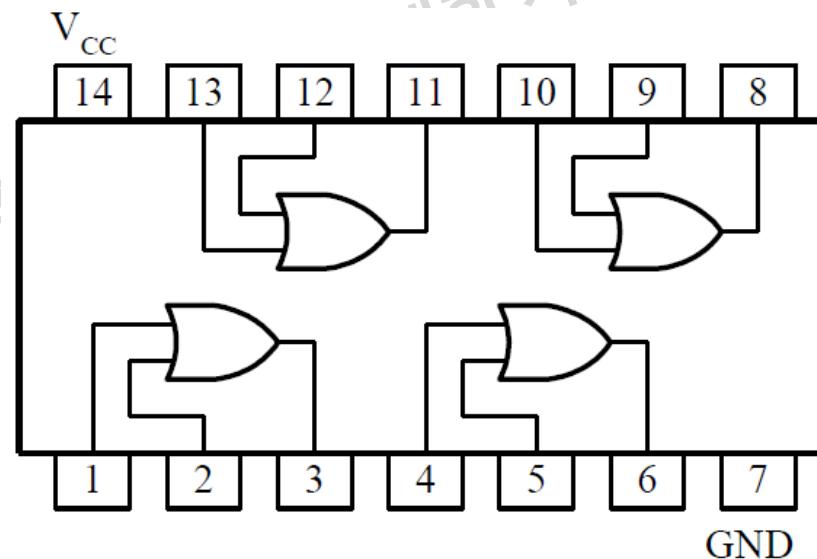
Hardware Description Language (HDL)

- HDL is an acronym of Hardware Description Language
- Two most commonly used HDLs:
 - Verilog HDL (also called Verilog for short)
 - VHDL (Very high-speed integrated circuits HDL)
- Features of HDLs:
 - Design can be described at a very abstract level.
 - Functional verification can be done early in the design cycle.
 - Designing with HDLs is analogous to computer programming.

Modules – Hardware Module Concept

- The basic unit of a digital system is a module.
- Each module consists of:
 - a **core** circuit (called **internal** or **body**) ---performs the required function
 - an **interface** (called **ports**) ---carries out the required communication between the core circuit and outside.

Mann
Dram



Modules –Verilog HDL modules

- module---The basic building block in Verilog HDL.
 - It can be an element or a collection of lower-level design blocks.

```
module Module name
    Port List, Port Declarations (if any)
    Parameters (if any)

    Declarations of wires, regs, and other variables

    Instantiation of lower level modules or primitives

    Data flow statements (assign)
    always and initial blocks. (all behavioral statements go
    into these blocks).

    Tasks and functions.

endmodule statement
```

Copyright 2022

Lexical Conventions

- Verilog HDL uses almost the same lexical conventions as C language.
 - Identifiers consists of alphanumeric characters, _, and \$.
 - Verilog is a case-sensitive language just like C.
- White space: blank space (`\b`), tabs (`\t`), and new line (`\n`).
- Comments:
 - `//` indicates that the remaining of the line is a comment.
 - `/* ... */` indicates what in between them are comments.

Lexical Conventions

- Sized number: <size>'<base format><number>
 - 4'b1001 ---a 4-bit binary number
 - 16'habcd ---a 16-bit hexadecimal number
- Unsized number: `<base format><number>
 - 2007 ---a 32-bit decimal number by default
 - `habc ---a 32-bit hexadecimal number
- x or z values: x denotes an unknown value; z denotes a high impedance value.
- Negative number: -<size>'<base format><number>
 - -4'b1001 ---a 4-bit binary number
 - -16'habcd ---a 16-bit hexadecimal number
- "_" and "?"
 - 16'b0101_1001_1110_0000
 - 8'b01??_11?? ---equivalent to a 8'b01zz_11zz

Lexical Conventions

- String: "Back to School and Have a Nice Semester"
- Coding style:
 - Use **lowercase letters** for all signal names, variable names, and port names.
 - Use **uppercase letters** for names of constants and user-defined types.
 - Use **meaningful names** for signals, ports, functions, and parameters.

The value set

- Four-value logic system in Verilog HDL
 - 0 and 1 represent logic values low and high, respectively.
 - z indicates the high-impedance condition of a node or net.
 - x indicates an unknown value of a net or node.

Value	Meaning
0	Logic 0, false condition
1	Logic 1, true condition
x	Unknown logic value
z	High impedance

Data types

- Verilog HDL has two classes of data types.
 - Nets mean any hardware connection points.
 - Variables represent any data storage elements.

Nets	Variables
wire	supply0
tri	supply1
wand	tri0
wor	tri1
triand	trireg
trior	

Data types

- A net variable
 - can be referenced anywhere in a module.
 - must be driven by a primitive, continuous assignment, force ... release, or module port.
- A variable
 - can be referenced anywhere in a module.
 - can be assigned value only within a procedural statement, task, or function.
 - cannot be an input or inout port in a module.

Port Declaration

- Port Declaration
 - input: input ports.
 - output: output ports.
 - inout: bidirectional ports
- Port Connection Rules
 - Named association
 - Positional association

```
Copyright 2022. 차세대반도체 혁신공유대학 사업단. All rights reserved.  
Copyright 2022. 차세대반도체 혁신공유대학 사업단. All rights reserved.  
  
module half_adder (x, y, s, c);  
    input x, y;  
    output s, c;  
    // -- half adder body-- //  
    // instantiate primitive gates  
    xor xor1 (s, x, y);  
    and and1 (c, x, y);  
endmodule  
  
module full_adder (x, y, cin, s, cout);  
    input x, y, cin;  
    output s, cout;  
    wire s1,c1,c2; // outputs of both half adders  
    // -- full adder body-- //  
    // instantiate the half adder  
    half_adder ha_1 (x, y, s1, c1);  
    half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));  
    or (cout, c1, c2);  
endmodule
```

Can only be connected by using positional association
Instance name is optional.

Connecting by using positional association
Connecting by using named association
Instance name is necessary.

Module Modeling Styles

- Structural style
- Dataflow style
- Behavioral or algorithmic style
- Mixed style
- In industry, RTL (register-transfer level) means
 - RTL = synthesizable behavioral + dataflow constructs

Structural modeling

- Structural style
 - Gate level comprises a set of interconnected gate primitives.
 - Switch level consists of a set of interconnected switch primitives.

```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
    input x, y;
    output s, c;
    // half adder body
    // instantiate primitive gates
    xor (s,x,y);
    and (c,x,y);
endmodule
```

```
// gate-level description of full adder
module full_adder (x, y, cin, s, cout);
    input x, y, cin;
    output s, cout;
    wire s1, c1, c2; // outputs of both half adders
    // full adder body
    // instantiate the half adder
    half_adder ha_1 (x, y, s1, c1);
    half_adder ha_2 (cin, s1, s, c2);
    or (cout, c1, c2);
endmodule
```

Hierarchical design

Dataflow modeling

- Dataflow style specifies the dataflow (i.e., data dependence) between registers.
- Use a set of continuous assignment statements
 - `assign [delay] l_value = expression`
 - delay: the amount of time between a change of operand used in expression and the assignment to l-value.
- **Continuous statement in a module execute concurrently regardless of the order they appear.**



```
module full_adder_dataflow(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
// specify the function of a full adder
assign #5 {c_out, sum} = x + y + c_in;
endmodule
```

↓
code is run
parallelly
like all links
at a time'

Behavioral modeling

- Use two procedural constructs: `initial` and `always`
- `initial` statement
 - Executed only once at simulation time 0
 - Used to set up the initial value of a variable
- `always` statement
 - Executed repeatedly
- At simulation time 0, both `initial` and `always` statements are executed concurrently.

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out; // sum and c_out need to be declared as reg types.
// specify the function of a full adder
always @(x, y, c_in) //or always @(x or y or c_in)
#5 {c_out, sum} = x + y + c_in;
endmodule
```

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input x, y, c_in;
output sum, c_out;
reg sum, c_out; // sum and c_out need to be declared as reg types.
// specify the function of a full adder
always @(*)
#5 {c_out, sum} = x + y + c_in;
endmodule
```

Mixed-Style Modeling

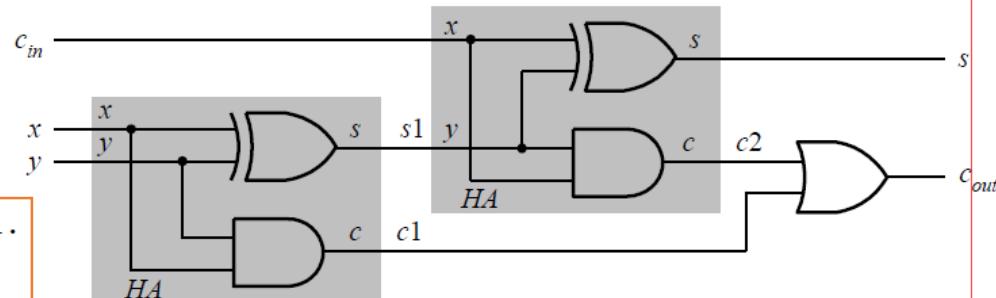
- Mixed style is the mixing use of above three modeling styles.
 - Commonly used in modeling large designs.

structural

dataflow

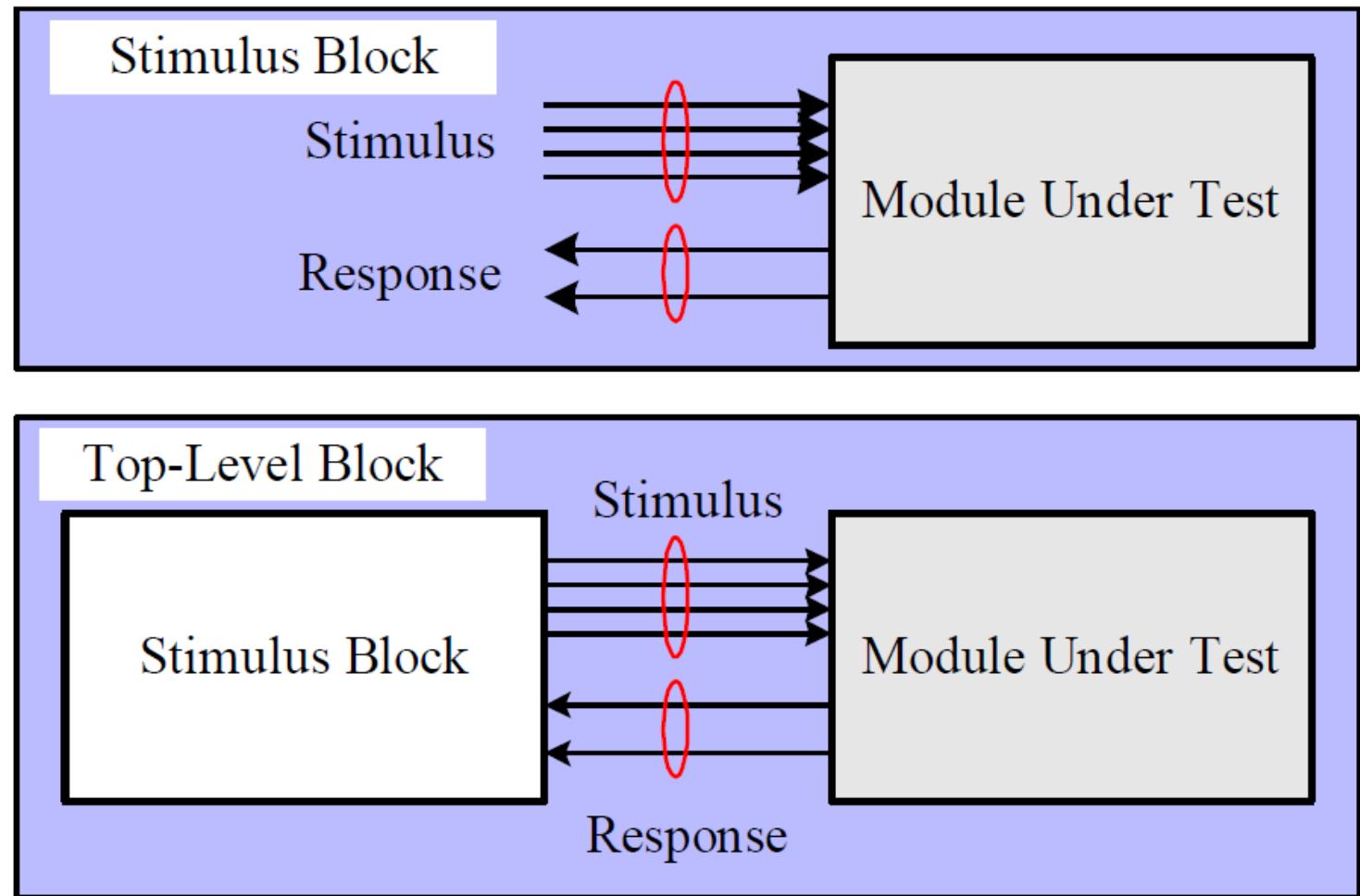
behavioral

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input x, y, c_in;
output s, c_out;
reg c_out;
wire s1, c1, c2;
// structural modeling of HA 1.
xor xor_ha1 (s1, x, y);
and and_ha1(c1, x, y);
// dataflow modeling of HA 2.
assign s = c_in ^ s1;
assign c2 = c_in & s1;
// behavioral modeling of output OR gate.
always @(*)
c_out = c1 | c2;
endmodule
```



Simulation

- Design
- Simulation
- Verification
- Stimulus block: testbench
- Unit under test (UUT)
- Design under test (DUT)



System Tasks for Simulation

- `$display` displays values of variables, string, or expressions
 - `$display(ep1, ep2, ..., epn);`
 - `ep1, ep2, ..., epn`: quoted strings, variables, expressions.
- `$monitor` monitors a signal when its value changes.
 - `$monitor(ep1, ep2, ..., epn);`
- `$monitoron` enables monitoring operation.
- `$monitoroff` disables monitoring operation.
- `$stop` suspends a simulation and enters an interactive debug mode.
- `$finish` terminates a simulation and exits the simulation process.

Time Scale for Simulations

- Time scale compiler directive
 - `timescale time_unit / time_precision
- The time_precision must not exceed the time_unit.
- Example:
 - with a timescale 1 ns/1 ps, the delay specification #15 corresponds to 15 ns.
- It uses the same time unit in both behavioral and gate-level modeling.
- For FPGA designs, it is suggested to use ns as the time unit.

Lecture plan

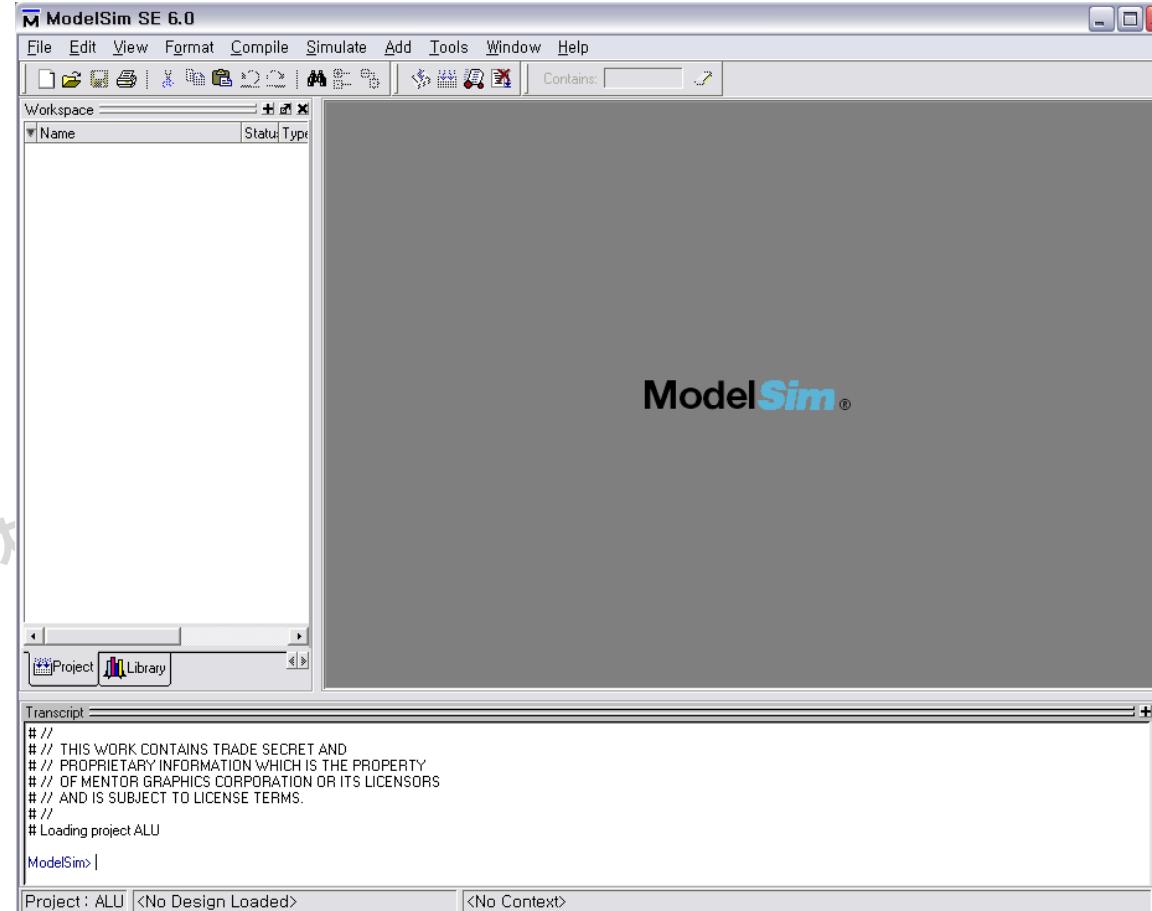
- Introduction to Verilog HDL

- Lab01: Hello ModelSim
- Lab02: Encoder
- Lab03: Up counter

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

About ModelSim

- ModelSim is a Verilog/VHDL simulation tool provided by Mentor graphics
 - Simulate digital logics
 - Use a GUI-based console



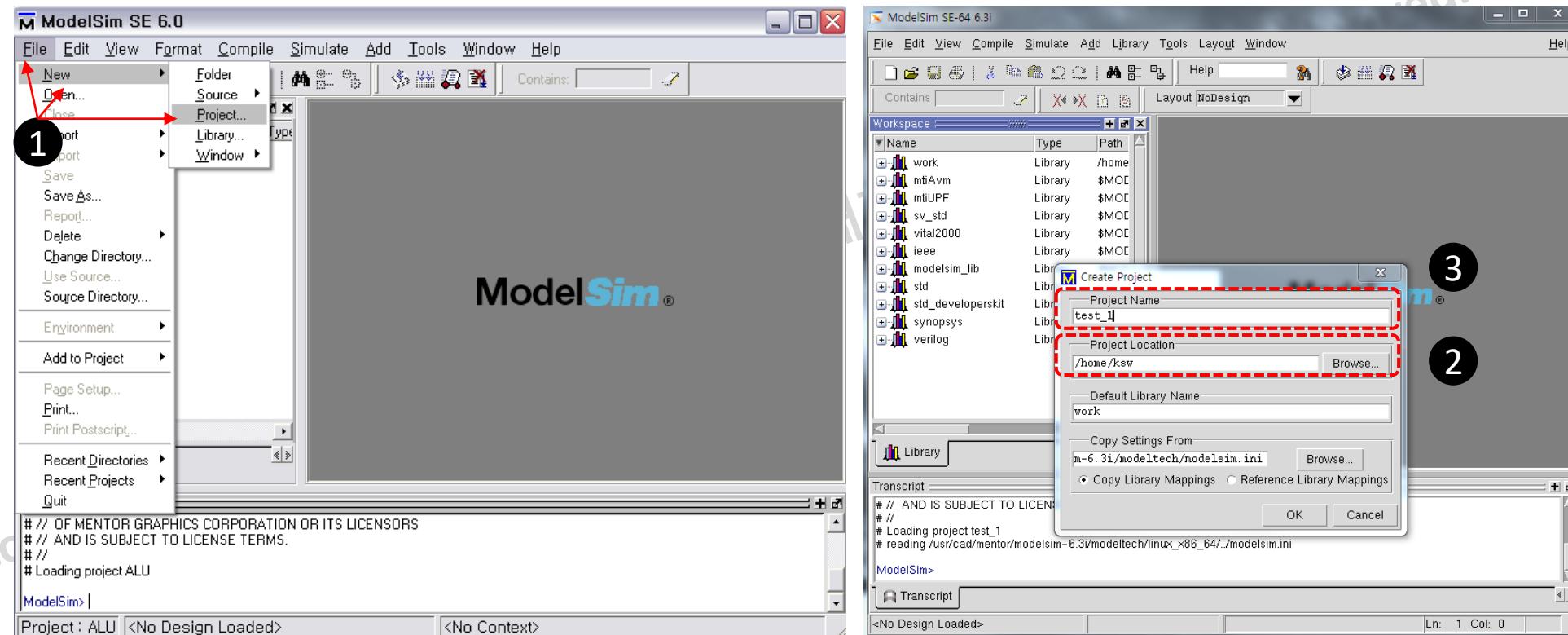
Lab 1: Hello ModelSim

- Lab 1:
 - Create your first project with ModelSim
 - Display "hello ModelSim!!!" on the command window

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

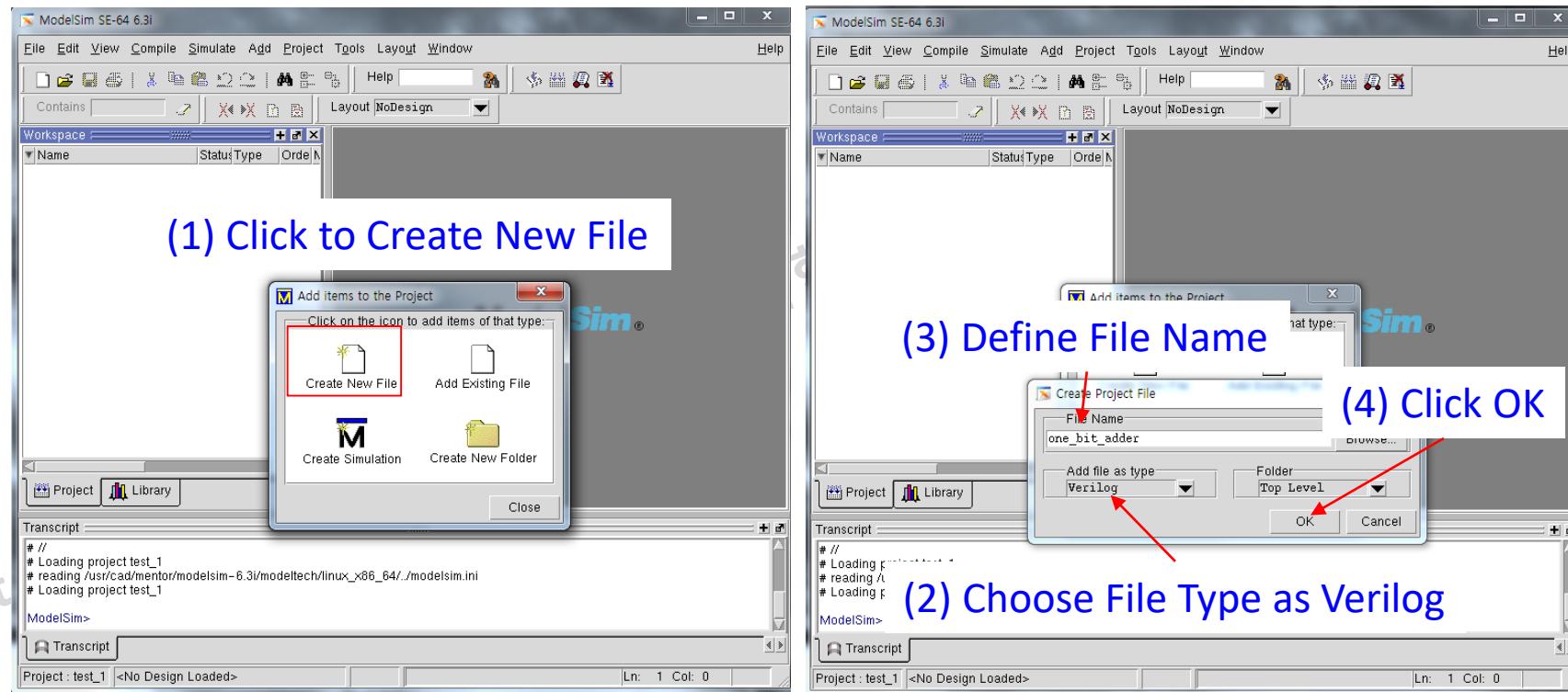
Create a new project

1. File → New → Project
2. Choose a directory or use a default directory
3. Define a project's name, i.e. "lab"



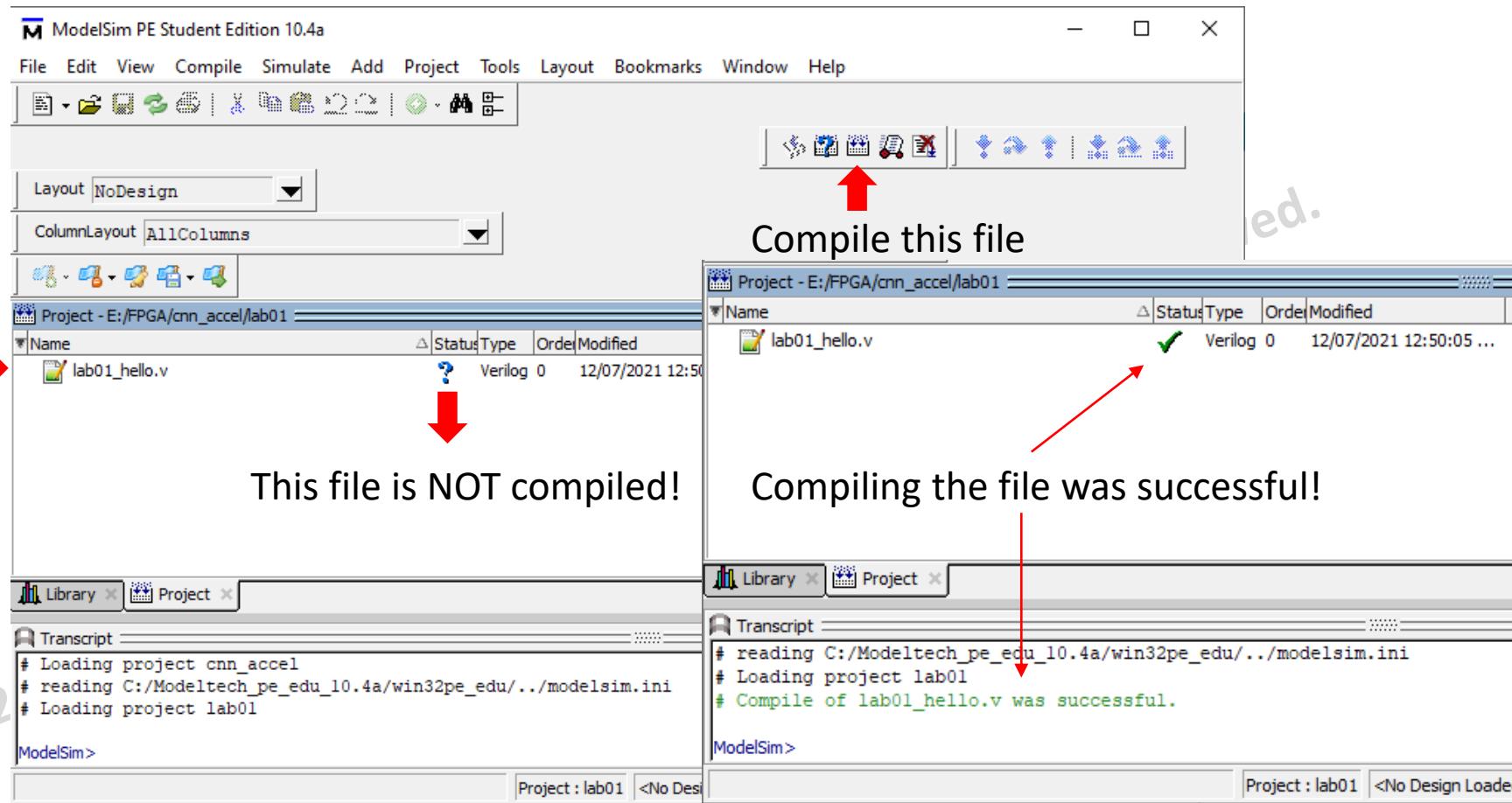
Create a new file

- After making a new project, to create a new file
 - A new file, lab01_hello.v, is created.



Compile

Create an empty file



- What does the compiling process do?

Coding

- Double click to the file
- Add the following code:

```
module lab1_hello;  
initial begin  
    $display("hello ModelSim!!!");  
end  
  
endmodule
```

Coding

- Coding syntaxes in Verilog

```
module lab1_hello; // Must end by a semicolon
initial begin
    $display("hello ModelSim!!!");
end
endmodule
```

(1) Define a module

(2) Name

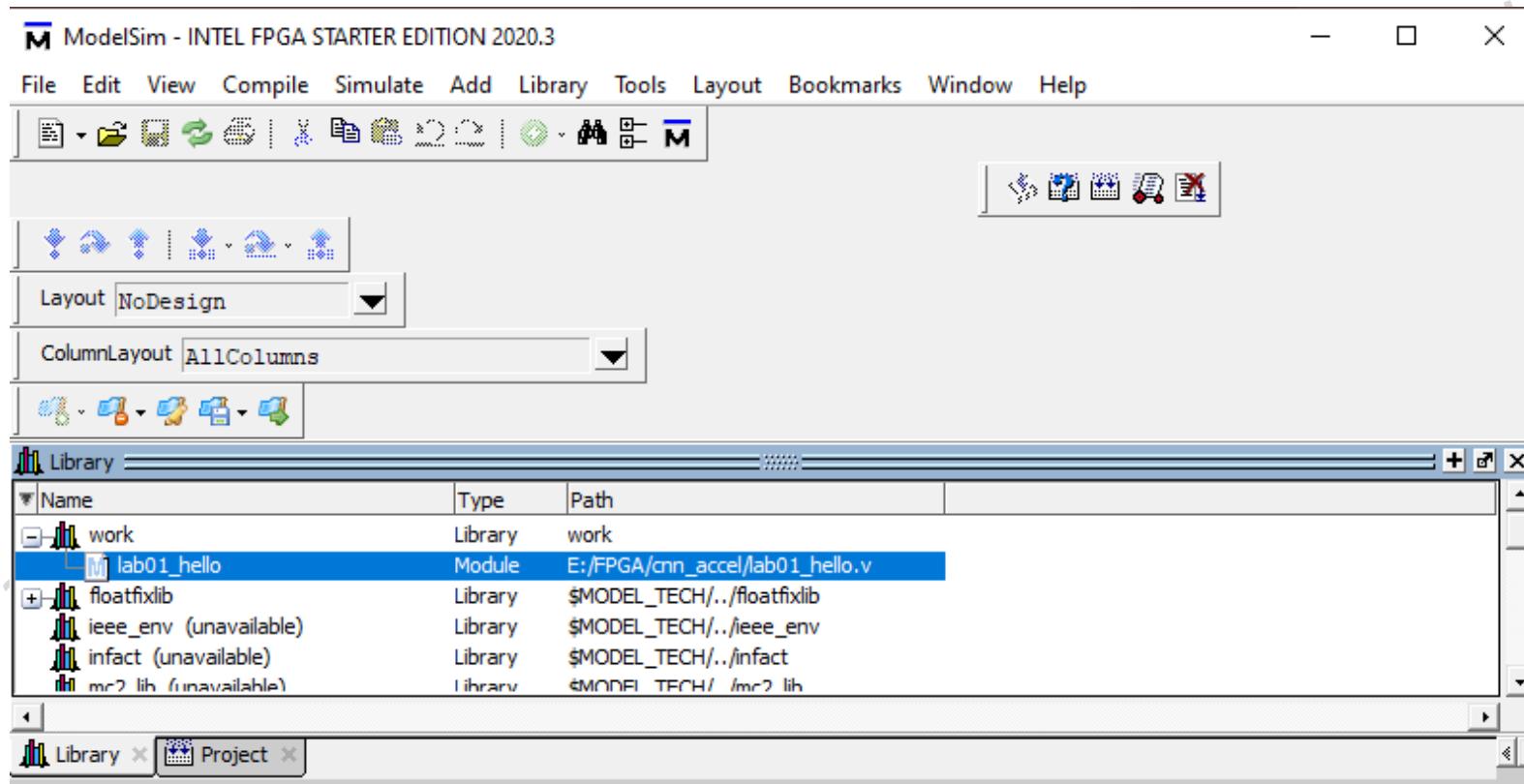
(3) Body of code -> Block
It uses a pair of "begin" and "end"

- Module definition

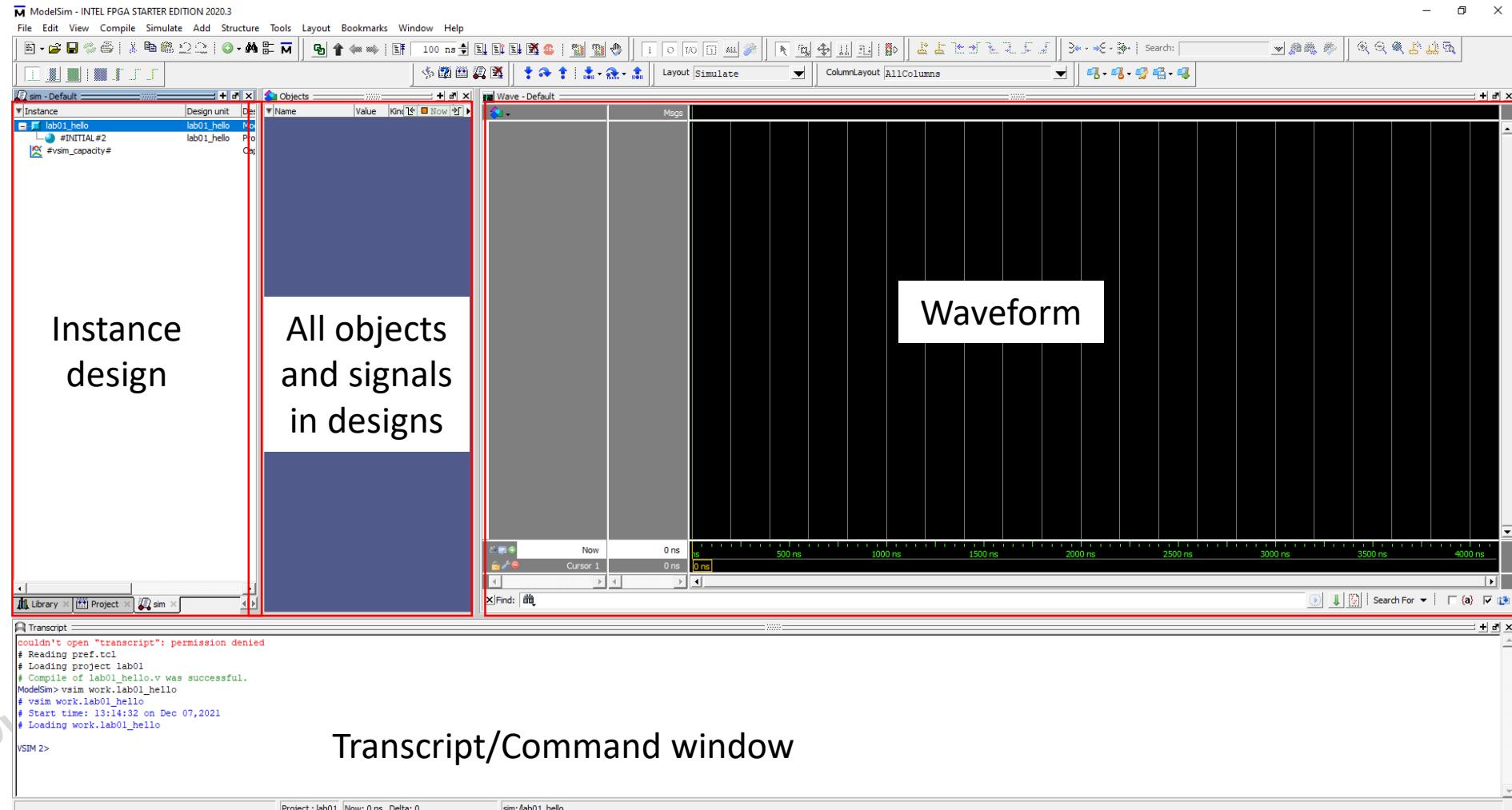
- Use a pair of "module" and "endmodule" to define a new module → Function
- Must end with a semicolon

Simulation

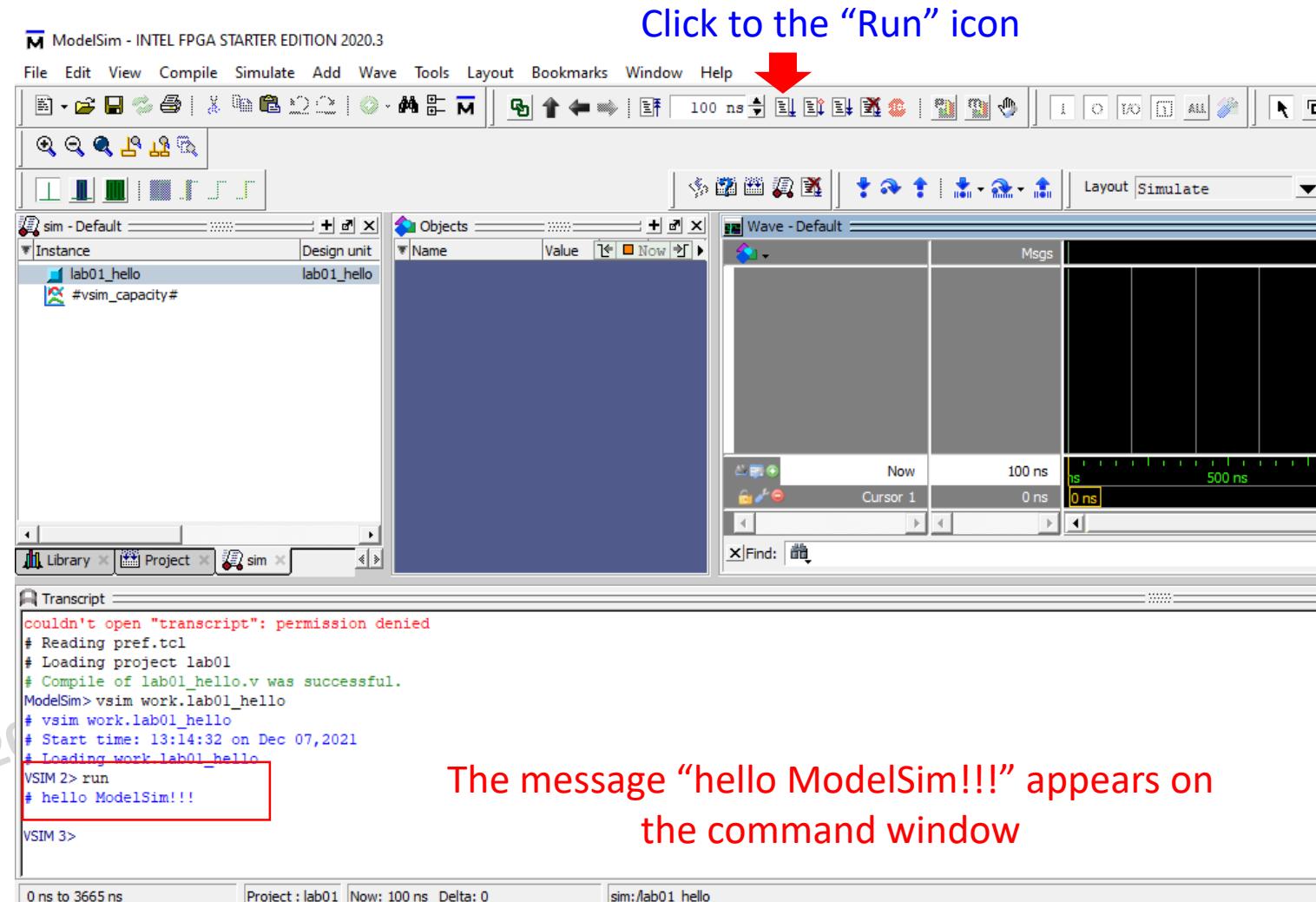
- After compiling the code successfully, we can check the new design
 - Library → work → lab01_hello
 - Right click to "lab01_hello" and select "Simulate"



Simulation environment

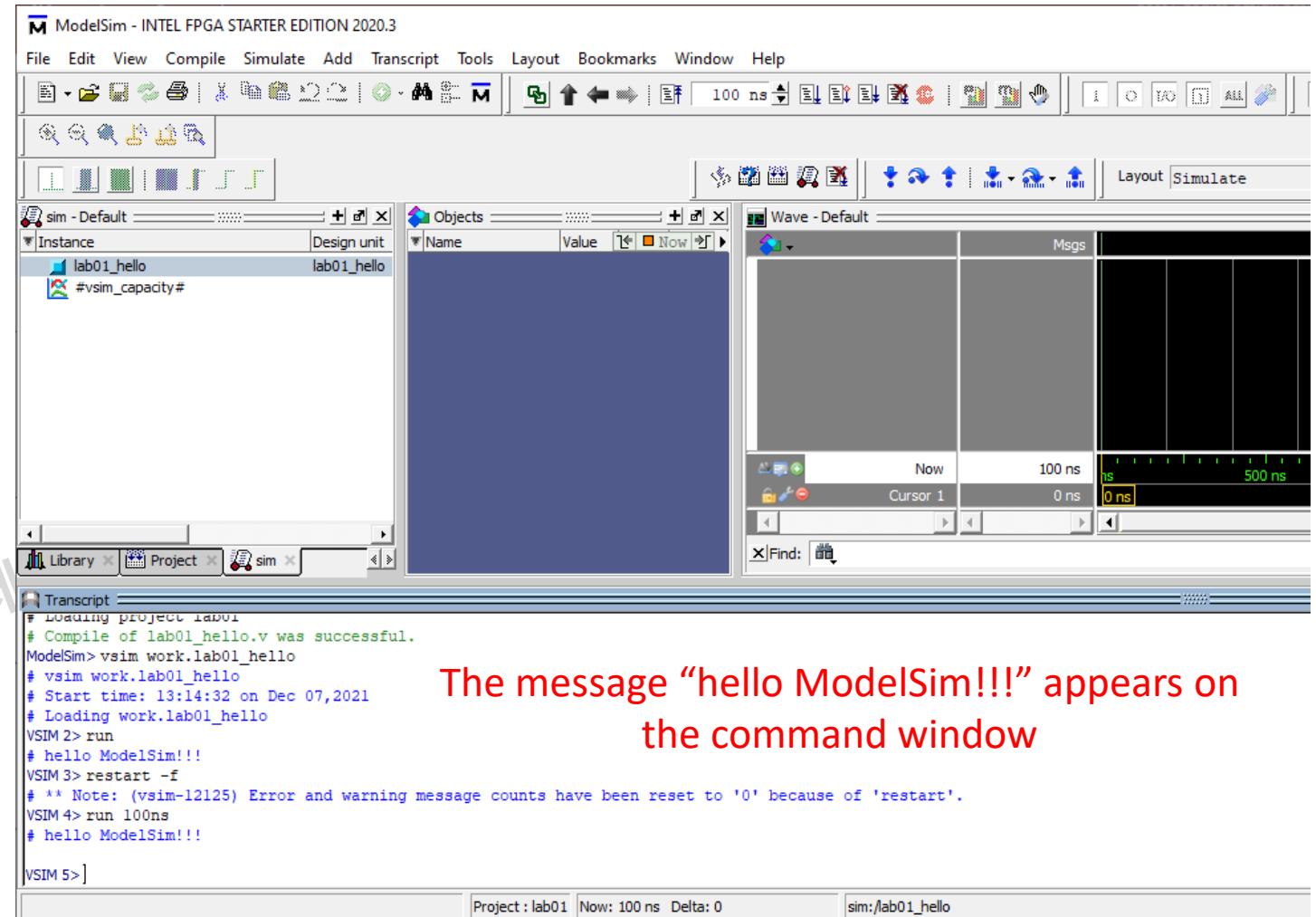


Simulation



Restart

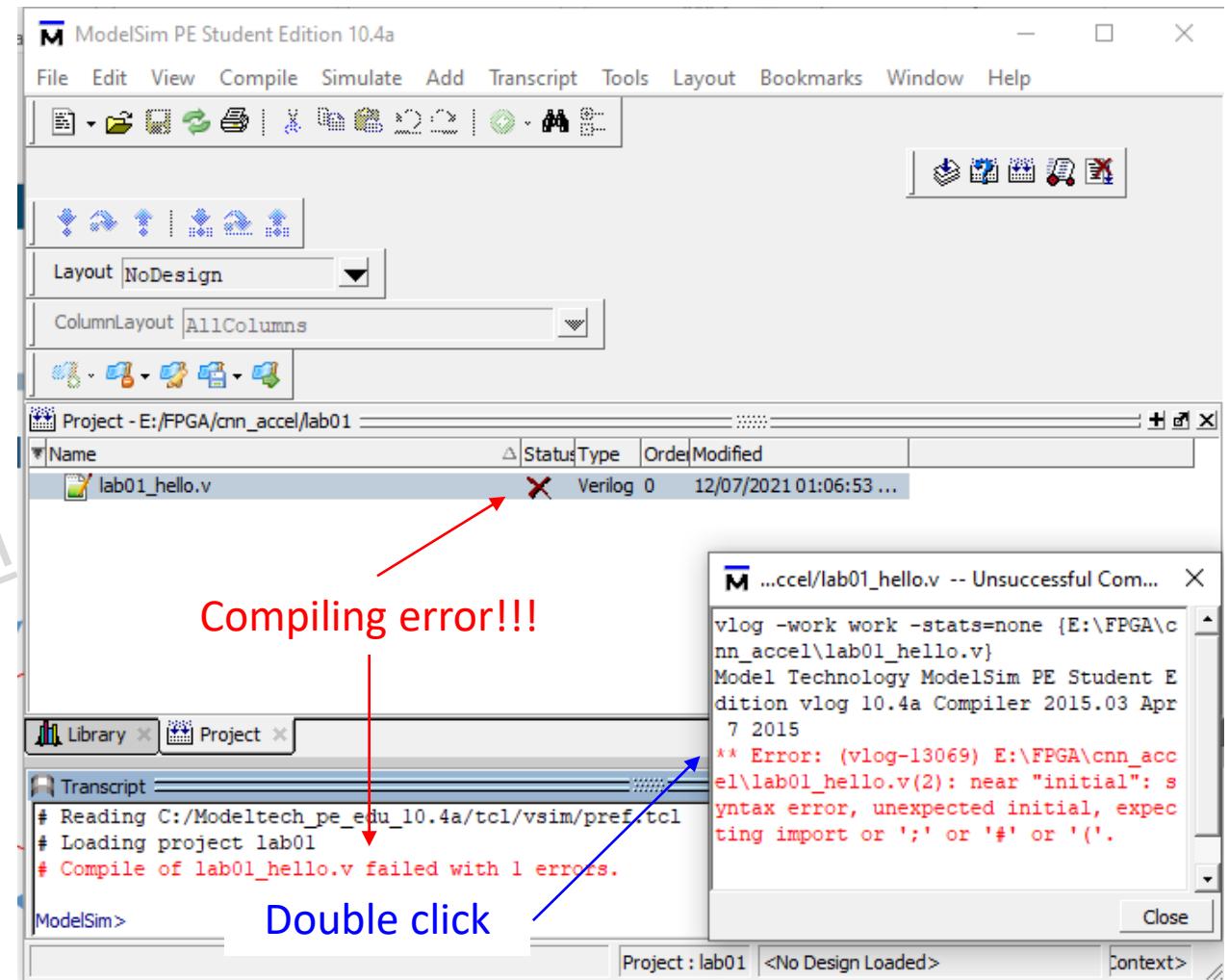
- We can use the command window to do simulation:
 - restart -f
 - run 100ns



Compiling error!!!

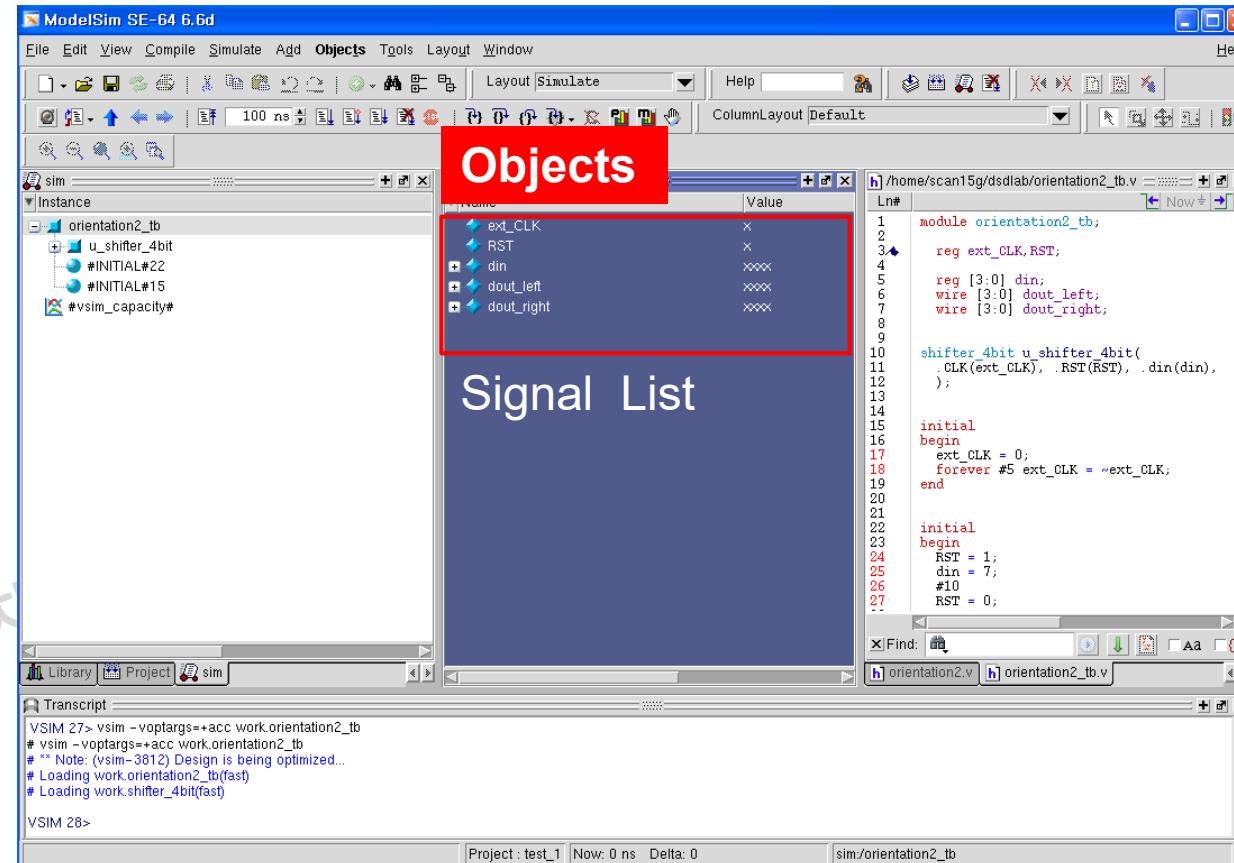
- In many cases, we make a mistake
 - Lead to an error when compiling the code
- For example, we forget a semicolon

```
module lab1_hello
initial begin
    $display("hello ModelSim!!!");
end
endmodule
```
- Double click to the line to check the popup error message in details
 - Syntax error near "initial", line 2 of lab01_hello.v



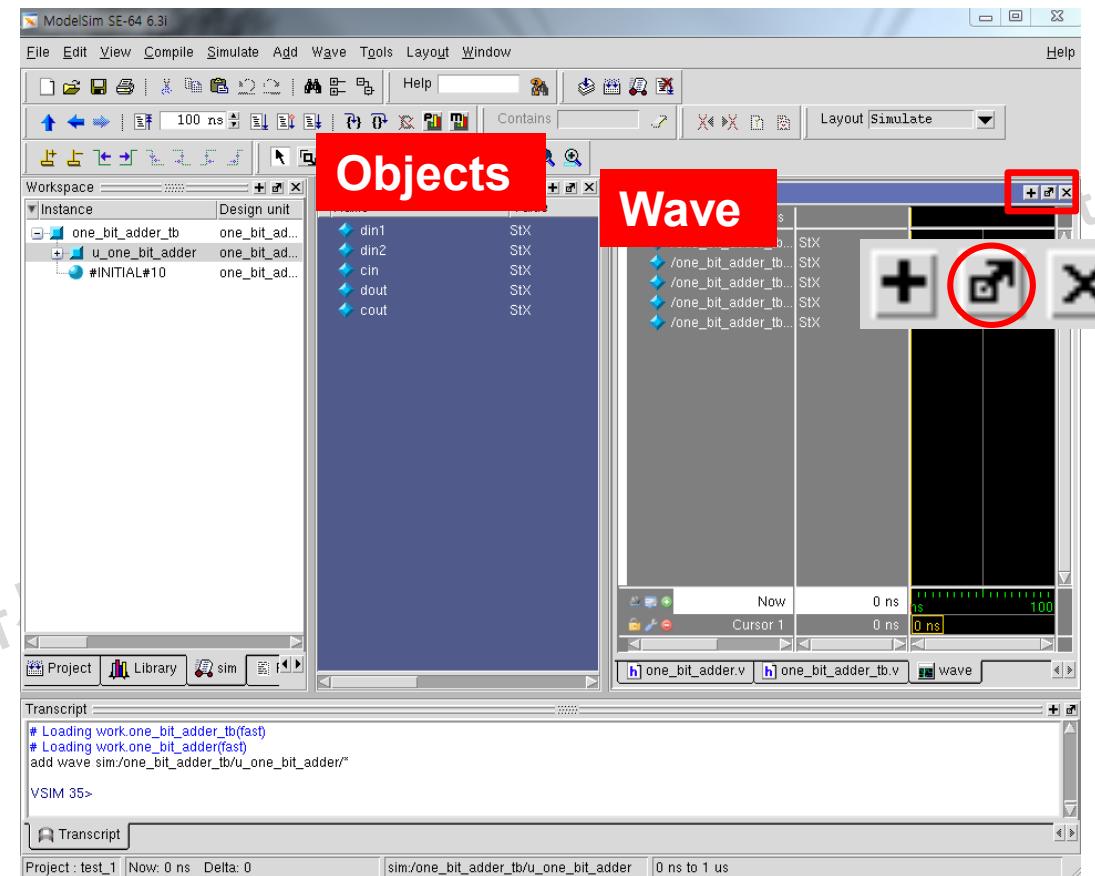
ModelSim - Simulation

- To see an instance Object, click View -> Objects
- To open waveform window, click View -> Wave

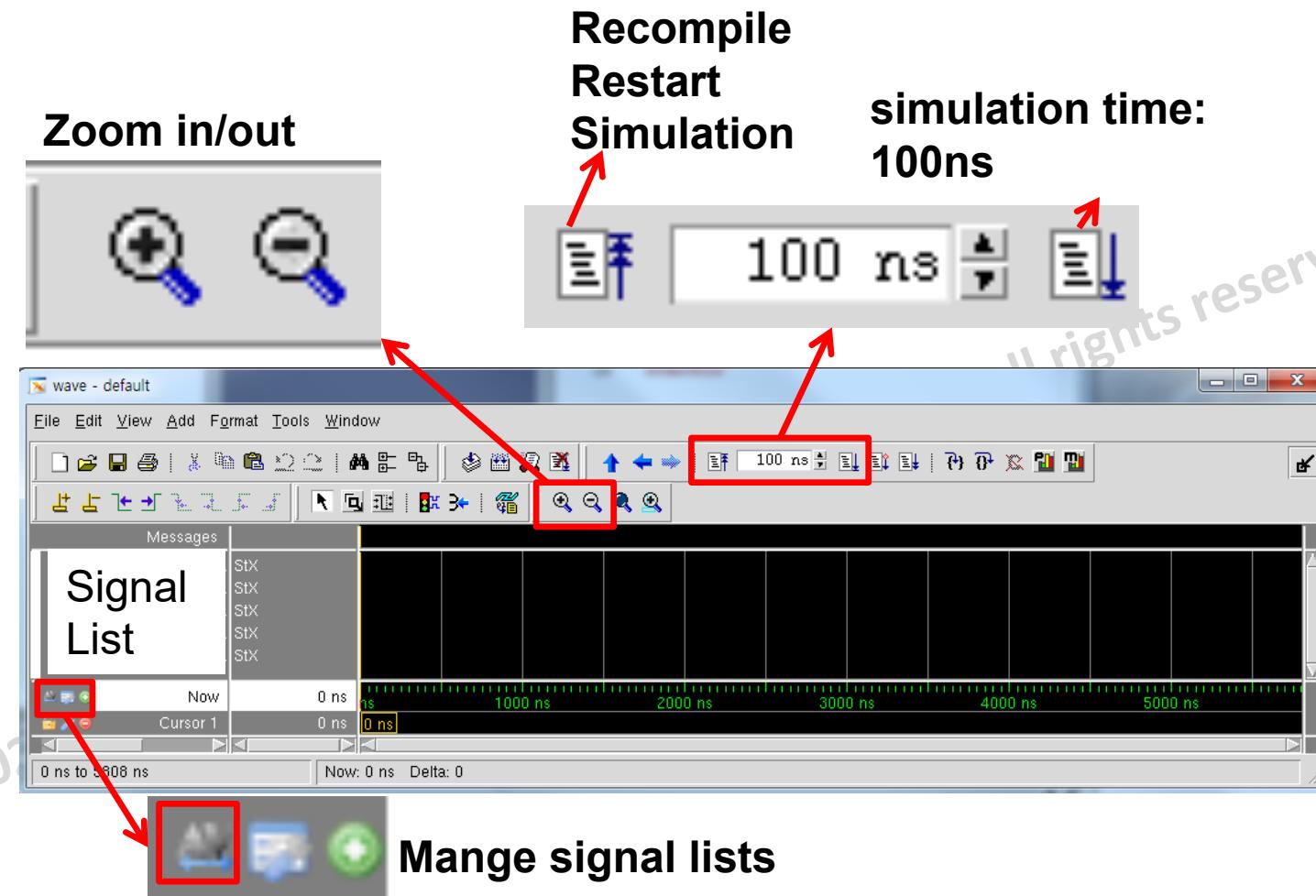


ModelSim - Simulation

- Drag and drop an object's signal to monitor its waveform



ModelSim - Simulation



Lecture plan

- Basics Logic Design
 - Motivation
 - Digit format
 - Digital logics
- Lab01: Hello ModelSim
- Lab02: Encoder
- Lab03: Counter

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

Combinatorial logic vs sequential logic

- Two types of circuits:
 - Combinatorial logic
 - Sequential logic

Combinational Logic

```
wire [1:0] a;  
assign a = 2'b00;
```

```
reg [1:0] b;  
always@*  
begin  
    b = 2'b00;  
end
```

Sequential Logic

```
reg [1:0] c;  
always@(posedge clk or negedge rstn)  
begin  
    c <= 2'b00;  
end
```

Lab 2: Encoder

- Lab 2: (Combinational logic) Design a four-to-two encoder
 - Implement an encoder module
 - Create a test bench
 - Run simulation
 - Show the output result

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

Encoder

- Design a 4-to-2 encoder.



- Input: din0, din1, din2, din3 => 1-bit values
- Output: dout => a 2-bit value
- The functionality or logic are described by a truth table

din0	din1	din2	din3	dout[1]	dout[0]
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Software reference

- Source code in C/Matlab
- The encoder can be considered as a function.
 - Define inputs and outputs
 - Describe the relationship between inputs and outputs.

```
int encoder4to2(int din0, int din1, int din2, int din3){  
    int dout;  
    if(din0==1 && din1==0 && din2==0 && din3==0) {  
        dout = 0;  
    }  
    else if(din0==0 && din1==1 && din2==0 && din3==0) {  
        dout = 1;  
    }  
    else if(din0==0 && din1==0 && din2==1 && din3==0) {  
        dout = 2;  
    }  
    else if(din0==0 && din1==0 && din2==0 && din3==1) {  
        dout = 3;  
    }  
    return dout;  
}
```

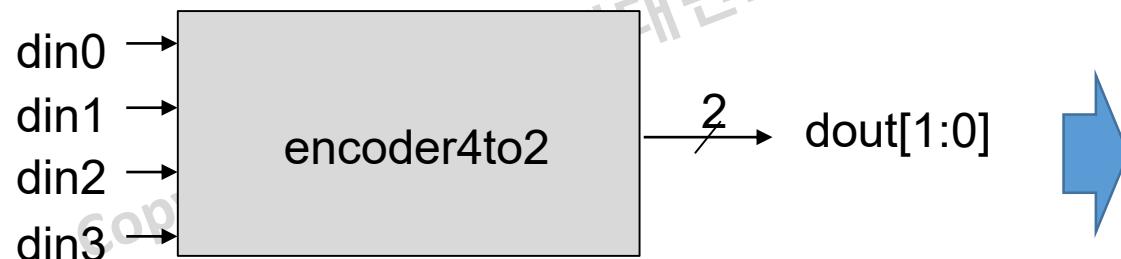
C code

```
function dout=encoder4to2(din0, din1, din2, din3)  
dout = 0; % Default value  
  
if(din0==1 && din1==0 && din2==0 && din3==0)  
    dout = 0;  
elseif (din0==0 && din1==1 && din2==0 && din3==0)  
    dout = 1;  
elseif(din0==0 && din1==0 && din2==1 && din3==0)  
    dout = 2;  
elseif(din0==0 && din1==0 && din2==0 && din3==1)  
    dout = 3;  
end  
end
```

MATLAB code

encoder4to2.v

- Create a new project "lab2_encoder"
- Create a new file "encoder4to2.v"
 - Input and output ports (~variables)
 - Input ports: din0, din1, din2, din3
 - Output port: dout
- Notes
 - We must indicate the number of bits for ports
 - Inputs are wires, while an output can be defined as a **register**
 - There is no integer, register, double variables in the design



Copyright 2022, 차세대반도체 혁신공유대학 사업단. All rights reserved.

```
module encoder4to2 ( din0,din1,din2,din3, dout );
    input  din0, din1, din2, din3;
    output reg [1:0] dout;
endmodule
```

The code is annotated with hand-drawn markings:

- A blue oval encloses the word "module".
- A red line highlights the identifier "encoder4to2".
- A green box highlights the output port declaration "dout".
- A purple box highlights the output port declaration "dout".
- A blue oval encloses the keyword "endmodule".
- A red curly brace groups the input declarations "din0, din1, din2, din3".
- A green curly brace groups the output declaration "dout".
- A handwritten note "array" is written next to the output declaration.

encoder4to2 (cont.)

- In Verilog, a code block is marked with "always"
 - It stays between "begin" and "end"
- Circuits
 - Combinatorial logic
 - always@*
 - Sequential logic (Later)
 - use a clock

```
module encoder4to2 ( din0,din1,din2,din3, dout );
    input  din0, din1, din2, din3;
    output reg [1:0] dout;

    always@*
    begin
        if(din0==1 && din1==0 && din2==0 && din3==0) begin
            dout = 2'b00;
        end
        else if(din0==0 && din1==1 && din2==0 && din3==0) begin
            dout = 2'b01;
        end
        else if(din0==0 && din1==0 && din2==1 && din3==0) begin
            dout = 2'b10;
        end
        else if(din0==0 && din1==0 && din2==0 && din3==1) begin
            dout = 2'b11;
        end
    end
endmodule
```

encoder4to2 (cont.)

- In Verilog, it is convenient to describe bit-level logics
 - Binary (b) format: $dout = 2'b00$ or, $dout = 2'b10$
 - Much easier than in C/Matlab
- We must be careful with Latch
 - 4 one-bit inputs => 16 cases
 - Only 4 cases are in Table
 - What is the output in others?

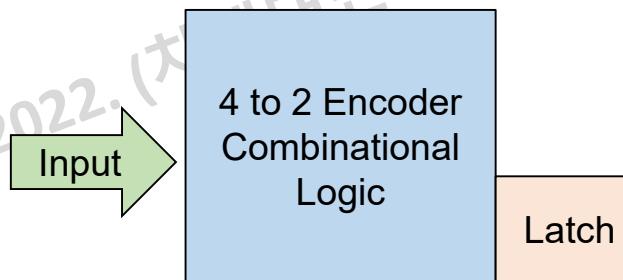
din0	din1	din2	din3	dout[1]	dout[0]
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

```
module encoder4to2 ( din0,din1,din2,din3, dout );
    input  din0, din1, din2, din3;
    output reg [1:0] dout;

    always@*
    begin
        if(din0==1 && din1==0 && din2==0 && din3==0) begin
            dout = 2'b00;
        end
        else if(din0==0 && din1==1 && din2==0 && din3==0) begin
            dout = 2'b01;
        end
        else if(din0==0 && din1==0 && din2==1 && din3==0) begin
            dout = 2'b10;
        end
        else if(din0==0 && din1==0 && din2==0 && din3==1) begin
            dout = 2'b11;
        end
    end
endmodule
```

encoder4to2 (cont.)

- Don't care in combinatorial logic
 - Latch: undefined condition
- To avoid "Latch" in combinatorial logic, it is necessary to add a "default" case
 - Here, dout = 2'b00 for any case that is not described in Table



```
always@*
begin
    if(din0==1 && din1==0 && din2==0 && din3==0) begin
        dout = 2'b00;
    end
    else if(din0==0 && din1==1 && din2==0 && din3==0) begin
        dout = 2'b01;
    end
    else if(din0==0 && din1==0 && din2==1 && din3==0) begin
        dout = 2'b10;
    end
    else if(din0==0 && din1==0 && din2==0 && din3==1) begin
        dout = 2'b11;
    end
    else dout = 2'b00;
end
```

Test bench

- How to test a module?
- In Verilog, we also need a test bench
 - Call a design module
 - Test it
 - Normal cases
 - Abnormal cases

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

Test bench

- Step 1: Define a module
 - Module name: encoder4to2_tb
- Step 2: Function call
 - 1) Define the signals: Registers connected to inputs of a test module, and wires for outputs.
 - 2) Call a test module (design under test (DUT)).
 - 3) Connect signals
 - 4) Syntax: do not forget "comma" and "semicolon"

```
reg din0, din1, din2, din3;  
wire [1:0] dout;  
encoder4to2 u_encoder4to2  
(  
.din0(din0),  
.din1(din1),  
.din2(din2),  
.din3(din3),  
.dout(dout)  
);
```

The diagram illustrates the four steps of defining a test bench module. Step 1 highlights the declaration of registers (din0, din1, din2, din3) and a wire (dout). Step 2 highlights the instantiation of the encoder4to2 module. Step 3 highlights the port connections (din0, din1, din2, din3, dout). Step 4 highlights the final closing parenthesis at the end of the module definition.

Test bench

- Step 3: Create test cases
 - Template: initial begin ... end
 - Each test case starts with a delay followed by an assignment of inputs
 - Normal cases
 - Test all cases in the Table
 - Exception cases
 - Test cases that do not exist in the Table

```
initial begin
#5      din0 = 1;
        din1 = 0;
        din2 = 0;
        din3 = 0;

#5      din0 = 0;
        din1 = 1;
        din2 = 0;
        din3 = 0;

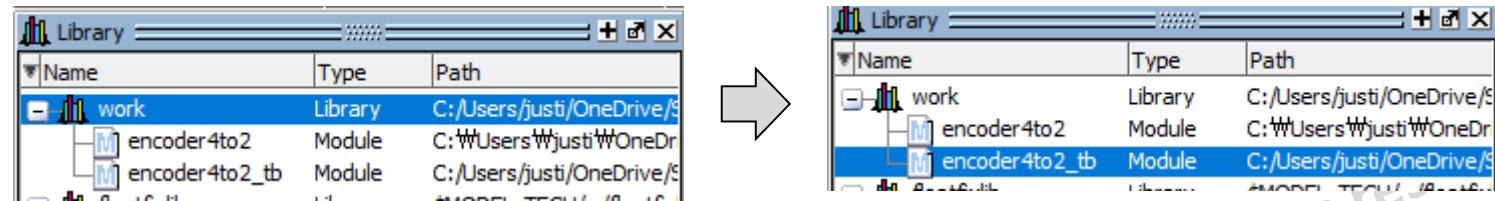
#5      din0 = 0;
        din1 = 0;
        din2 = 1;
        din3 = 0;

#5      din0 = 0;
        din1 = 0;
        din2 = 0;
        din3 = 1;

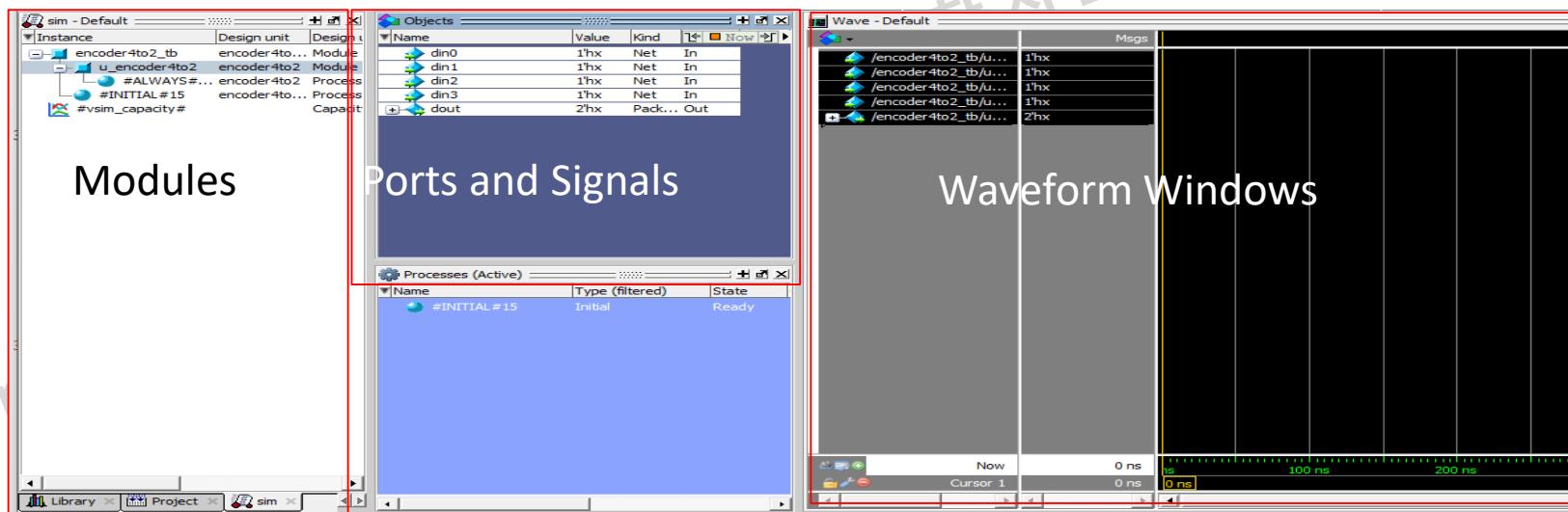
#5      din0 = 0;
        din1 = 0;
        din2 = 0;
        din3 = 0;
end
```

Simulation

- After all files are successfully compiled, all design modules can be seen at the tab Library>work

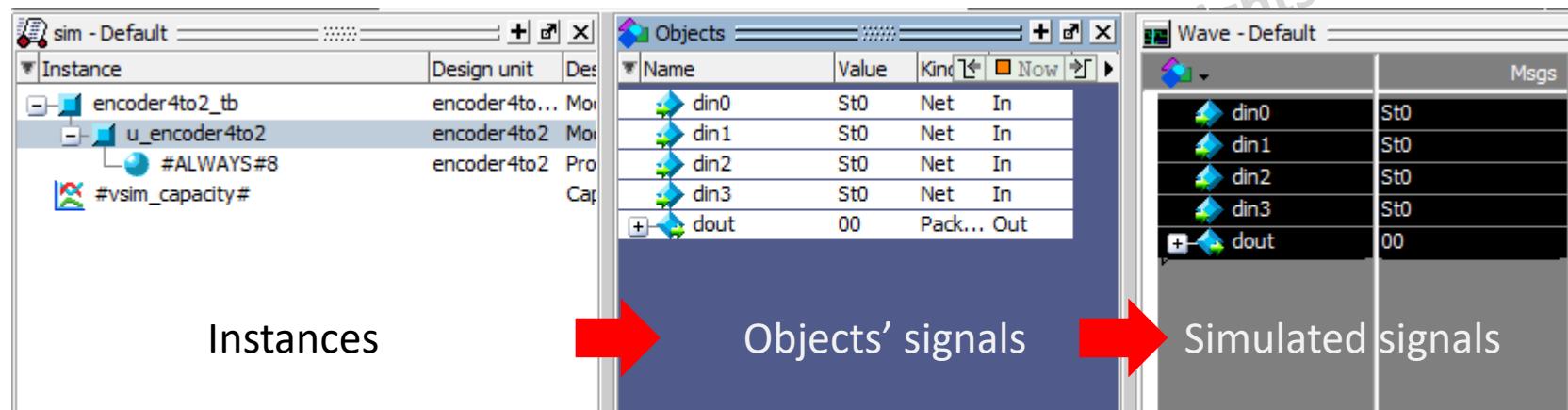


- Waveform: Select a module >> right click and choose simulate



Waveform

- Select signals, then right click and choose "Add Wave"
- You also can use drag and drop



Simulation

- Test case 1

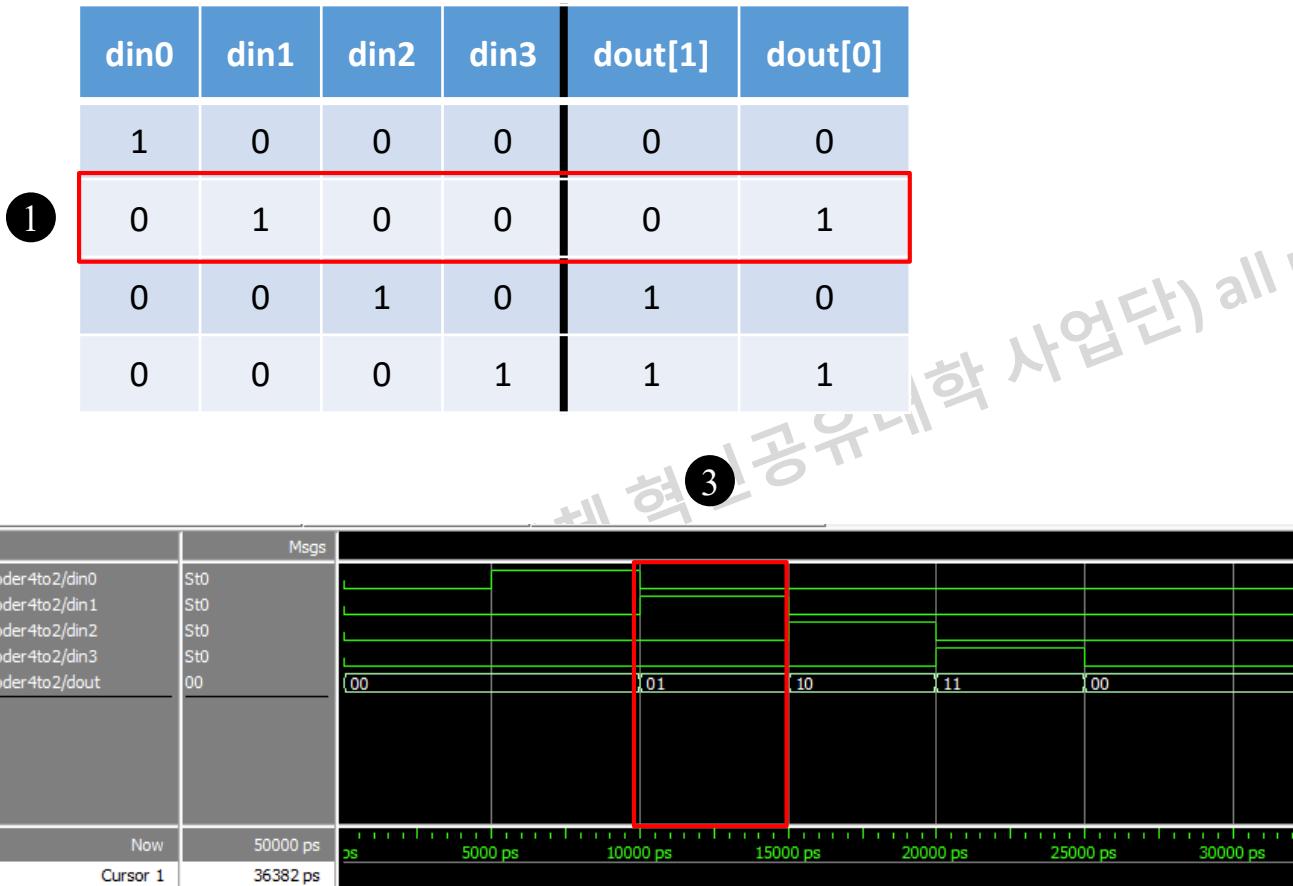
din0	din1	din2	din3	dout[1]	dout[0]
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1



```
initial begin
    #5      din0 = 1;
            din1 = 0;
            din2 = 0;
            din3 = 0;
    #5      din0 = 0;
            din1 = 1;
            din2 = 0;
            din3 = 0;
    #5      din0 = 0;
            din1 = 0;
            din2 = 1;
            din3 = 0;
    #5      din0 = 0;
            din1 = 0;
            din2 = 0;
            din3 = 1;
    #5      din0 = 0;
            din1 = 0;
            din2 = 0;
            din3 = 0;
end
```

Simulation

- Test case 2



```
initial begin
#5      din0 = 1;
din1 = 0;
din2 = 0;
din3 = 0;

#5      din0 = 0;
din1 = 1;
din2 = 0;
din3 = 0;

#5      din0 = 0;
din1 = 0;
din2 = 1;
din3 = 0;

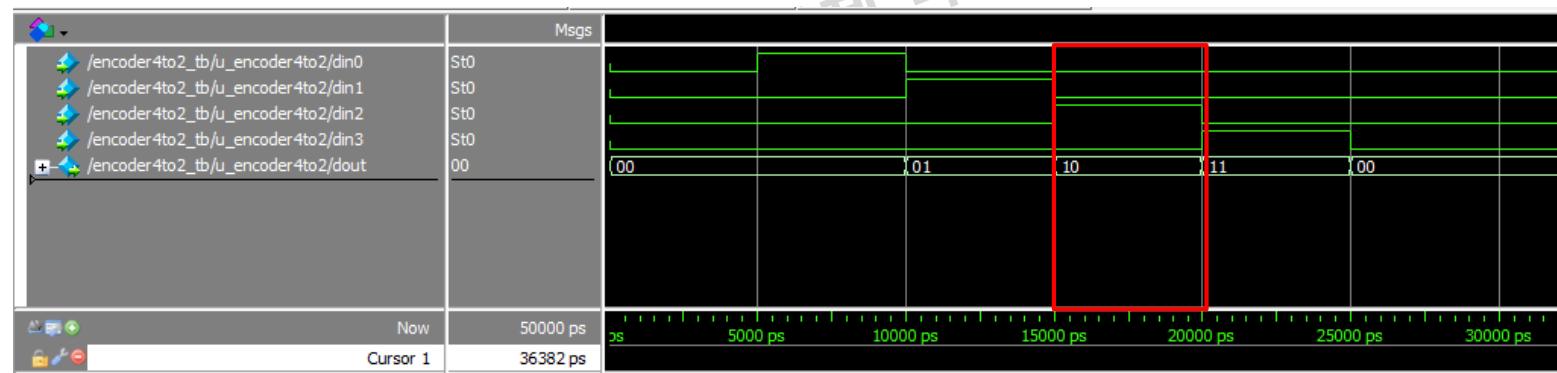
#5      din0 = 0;
din1 = 0;
din2 = 0;
din3 = 1;

#5      din0 = 0;
din1 = 0;
din2 = 0;
din3 = 0;
end
```

Simulation

- Test case 3

din0	din1	din2	din3	dout[1]	dout[0]
1	0	0	0	0	0
0	1	0	0	0	1
1	0	1	0	1	0
0	0	0	1	1	1

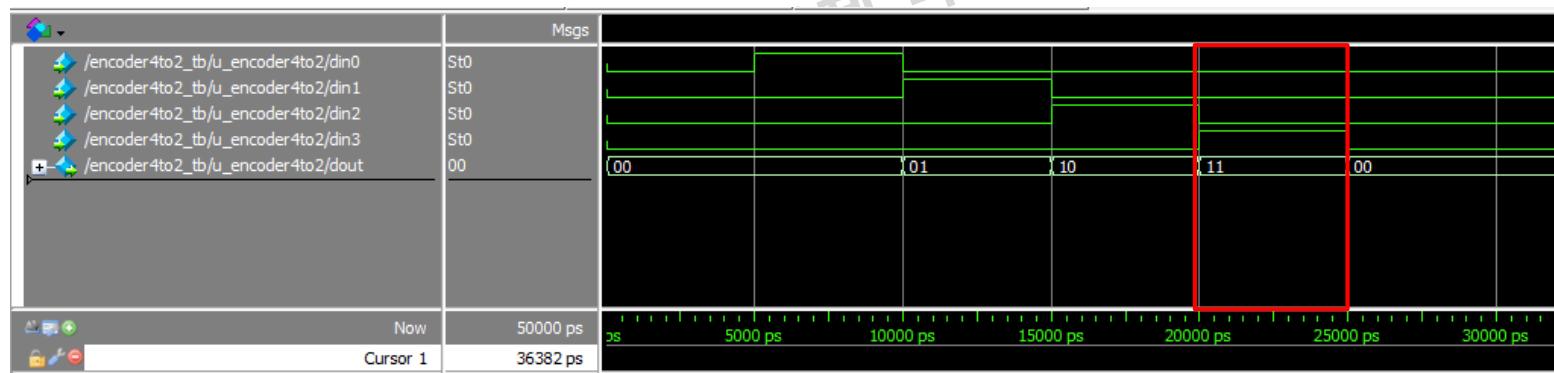


```
initial begin
#5      din0 = 1;
        din1 = 0;
        din2 = 0;
        din3 = 0;
#5      din0 = 0;
        din1 = 1;
        din2 = 0;
        din3 = 0;
#5      din0 = 0;
        din1 = 0;
        din2 = 1;
        din3 = 0;
#5      din0 = 0;
        din1 = 0;
        din2 = 0;
        din3 = 1;
#5      din0 = 0;
        din1 = 0;
        din2 = 0;
        din3 = 0;
end
```

Simulation

- Test case 4

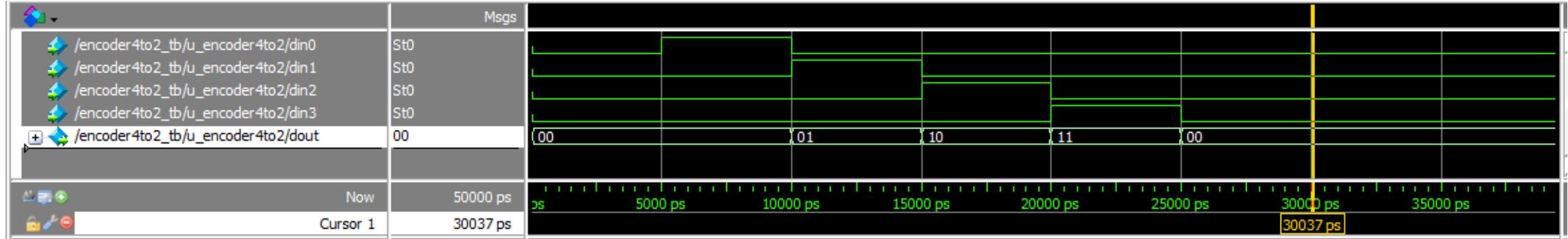
din0	din1	din2	din3	dout[1]	dout[0]
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
1	0	0	1	1	1



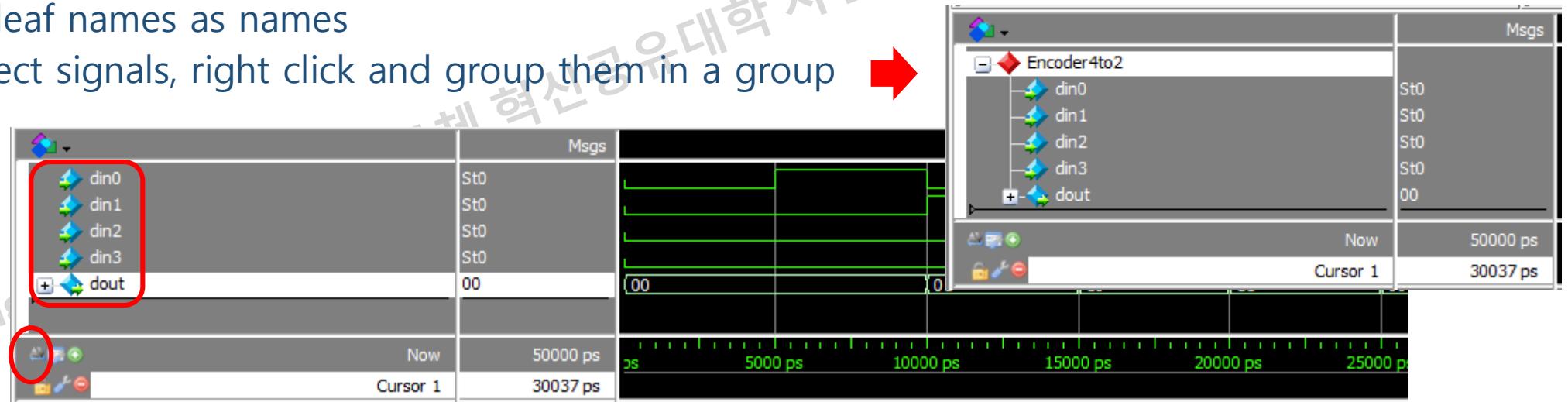
```
initial begin
#5      din0 = 1;
        din1 = 0;
        din2 = 0;
        din3 = 0;
#5      din0 = 0;
        din1 = 1;
        din2 = 0;
        din3 = 0;
#5      din0 = 0;
        din1 = 0;
        din2 = 1;
        din3 = 0;
#5      din0 = 0;
        din1 = 0;
        din2 = 0;
        din3 = 1;
#5      din0 = 0;
        din1 = 0;
        din2 = 0;
        din3 = 0;
end
```

Name

- Names are too long since they include modules and sub modules



- Toggle leaf names as names
 - Select signals, right click and group them in a group



To do ...

- Implement encoder4to2.v and encoder4to2_tb.v by completing the missing codes
- Do simulation with time = 30ns
- Show the output waveform

```
module encoder4to2(din0, din1, din2, din3, dout);
    input din0, din1, din2, din3;
    output reg[1:0] dout;
    always@*
    begin
        // Normal cases. Check Table in the PPT file
        if(din0==1 && din1==0 && din2==0 && din3==0) begin
            dout = 2'b00;
        end
        else if(din0==0 && din1==1 && din2==0 && din3==0) begin
            dout = 2'b01;
        end
        else if(din0==0 && din1==0 && din2==1 && din3==0) begin
            dout = 2'b10;
        end
        // Insert your code
        //{{{
        // Case 4:
        // Undefined cases/Default for handling latch
        //}}}
    end
endmodule
```

encoder4to2.v

```
// Delay 5 ns
// Test case 3
#5 din0 = 0;
din1 = 0;
din2 = 1;
din3 = 0;

// Delay 5 ns
// Test case 4: Insert your code
//{{{
//}}}

// Delay 5 ns
#5 din0 = 0;
din1 = 0;
din2 = 0;
din3 = 0;

end

endmodule
```

encoder4to2_tb.v

Lecture plan

- Basics Logic Design
 - Motivation
 - Digit format
 - Digital logics
- Lab01: Hello ModelSim
- Lab02: Encoder
- Lab03: Counter

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

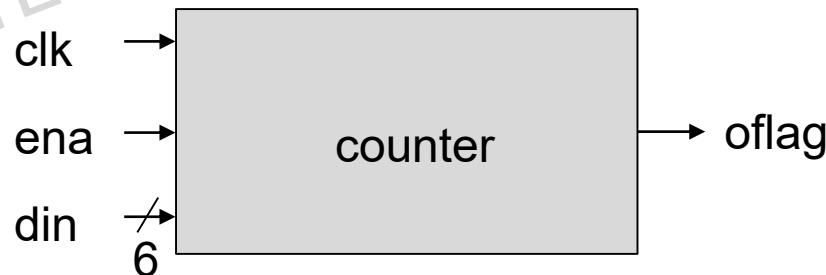
Lab 3: Counter

- Lab 3: (Sequential logic) Design a counter
 - Implement a counter module
 - Create a test bench
 - Run simulation
 - Show the output result



counter.v

- Create a new project "lab3_counter"
- Create a new file, "counter.v"
 - Inputs:
 - clk: clock signals
 - din[5:0]: input value
 - ena: enable signal
 - Internal counter: cnt[5:0]
 - Output
 - oflag: raise HIGH when an internal counter goes to ZERO.



counter.v: Pseudocode

- Internal counter (cnt)
 - If 'cnt' != 0
 - Decrease by 1
 - Else
 - If 'ena' becomes 1
 - Update 'cnt'
- Output
 - If 'cnt' is 1
 - Assign 'oflag' to 1
 - Otherwise
 - Set it to 0
- Hint: Use the sequential logic

Copyright 2022. (차세대반도체 혁신공유대학 사업단) all rights reserved.

The diagram shows a code snippet for sequential logic:

```
Sequential Logic
always@(posedge clk)
begin
end
```

A red circle highlights the `always@(posedge clk)` statement. A red arrow points from the circled text to the number '2' located above the code block.

Test bench

- counter_tb.v
 - Signals
 - Counter design under test
 - Clock module
 - Test case

```
'define CLOCK_PERIOD 10          //100MHz
module counter_tb;
    reg clk;
    reg [5:0] din;
    reg ena;
    wire oflag;

    // Counter design under test
    counter u_counter(.clk(clk), .din(din), .ena(ena), .oflag(oflag));

    // Clock
    initial begin
        clk = 0;
        forever #(CLOCK_PERIOD/2) clk = ~clk;
    end

    // Test case
    initial begin
        din = 6'd0;
        ena = 1'd0;

        // Count from 8 to 0
        #(CLOCK_PERIOD * 5)           din = 6'd8;
                                         ena = 1'd1;
                                         #(CLOCK_PERIOD)      ena = 1'd0;

        // Count from 16 to 0
        #(CLOCK_PERIOD * 30)          din = 6'd16;
                                         ena = 1'd1;
                                         #(CLOCK_PERIOD)      ena = 1'd0;
    end
endmodule
```

Test bench: Clock

- Pseudocode
 - Clock is initialized at 0 (or 1).
 - Every half of cycles
 - Clock changes the value
 - 1 to 0 => Falling edge
 - 0 to 1 => Rise edge

```
// Clock
initial begin
    clk = 0;
    forever #(CLOCK_PERIOD/2) clk = ~clk;
end
```

counter_tb.v

```
// Sequential logic
always@(posedge clk) begin
    .....
end
```

counter.v

Test case

- Two test cases
 - Counting from 8 to 0
 - Counting from 16 to 0

```
initial begin
    din = 6'd0;
    ena = 1'd0;

    // Count from 8 to 0
    #(`CLOCK_PERIOD * 5)
    din = 6'd8;
    ena = 1'd1;
    ena = 1'd0;

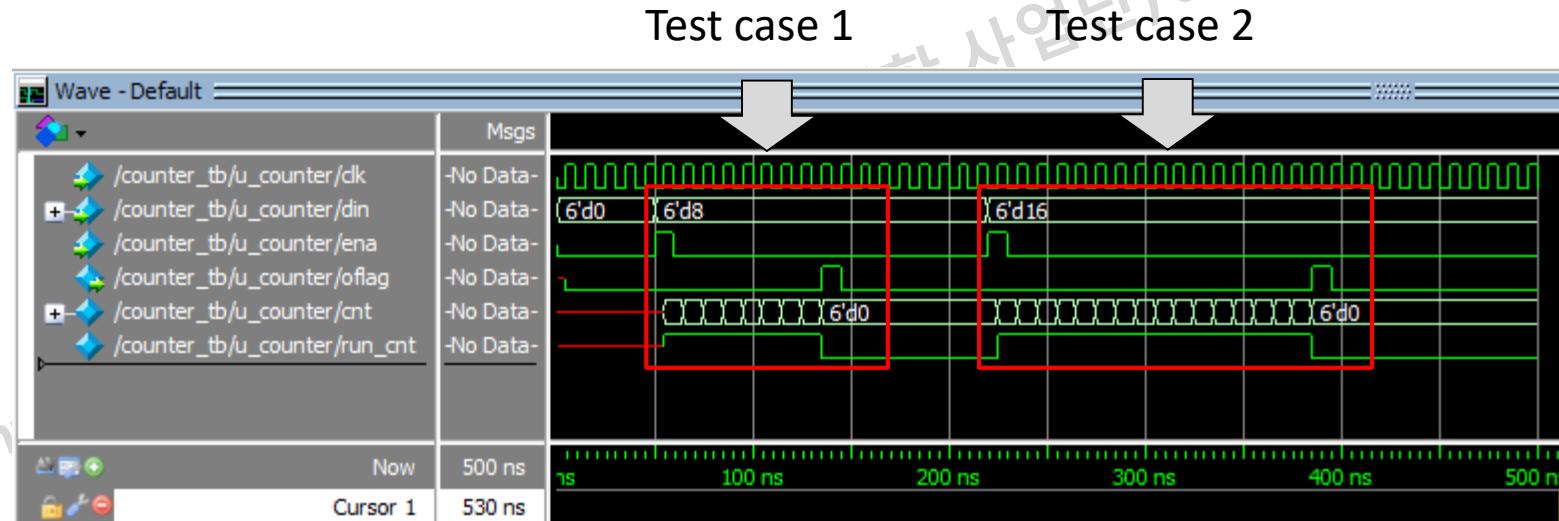
    // Count from 16 to 0
    #(`CLOCK_PERIOD * 16)
    din = 6'd16;
    ena = 1'd1;
    ena = 1'd0;

end
```

- ➊ Initialize values
 - Set the ‘enable’ signal to HIGH
 - Set an input
- ➋ Set the ‘enable’ signal to LOW
- ➌ Set the ‘enable’ signal to HIGH
 - Set other input
- ➍ Set the ‘enable’ signal to LOW

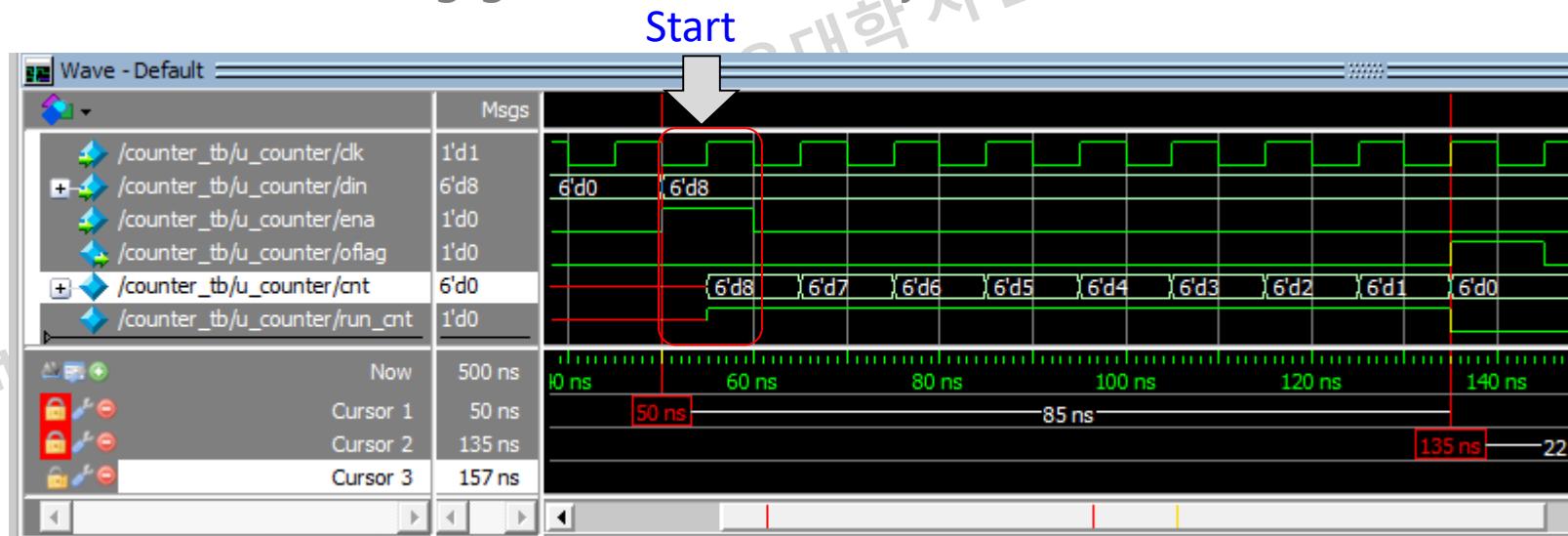
Waveform

- Signals:
 - Input/output: clk, din, ena, oflag
 - Internal: cnt
- Two test cases
- Simulation time: 500 ns



Waveform

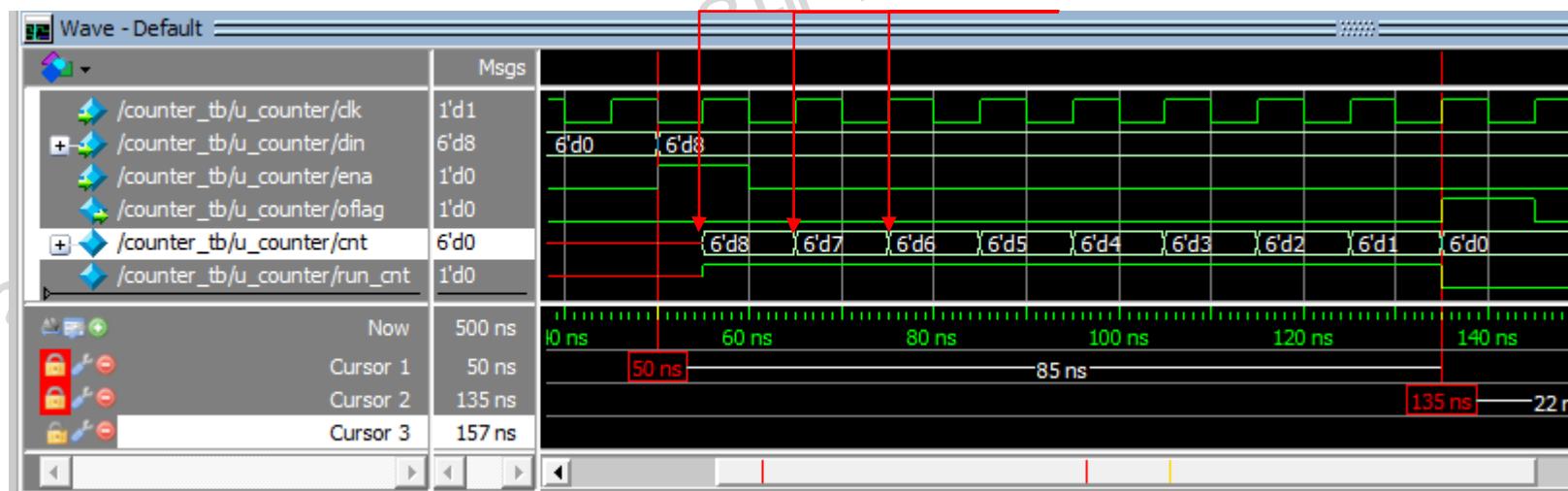
- Start
 - 'ena' goes HIGH and keeps in one cycle
 - A new value is written to 'din'
- Counting
 - At rising (positive) edge of 'clk', internal 'cnt' deceases by 1 if possible
- Output
 - 'cnt' goes from 1 to 0, oflag goes HIGH in one cycle.



Waveform

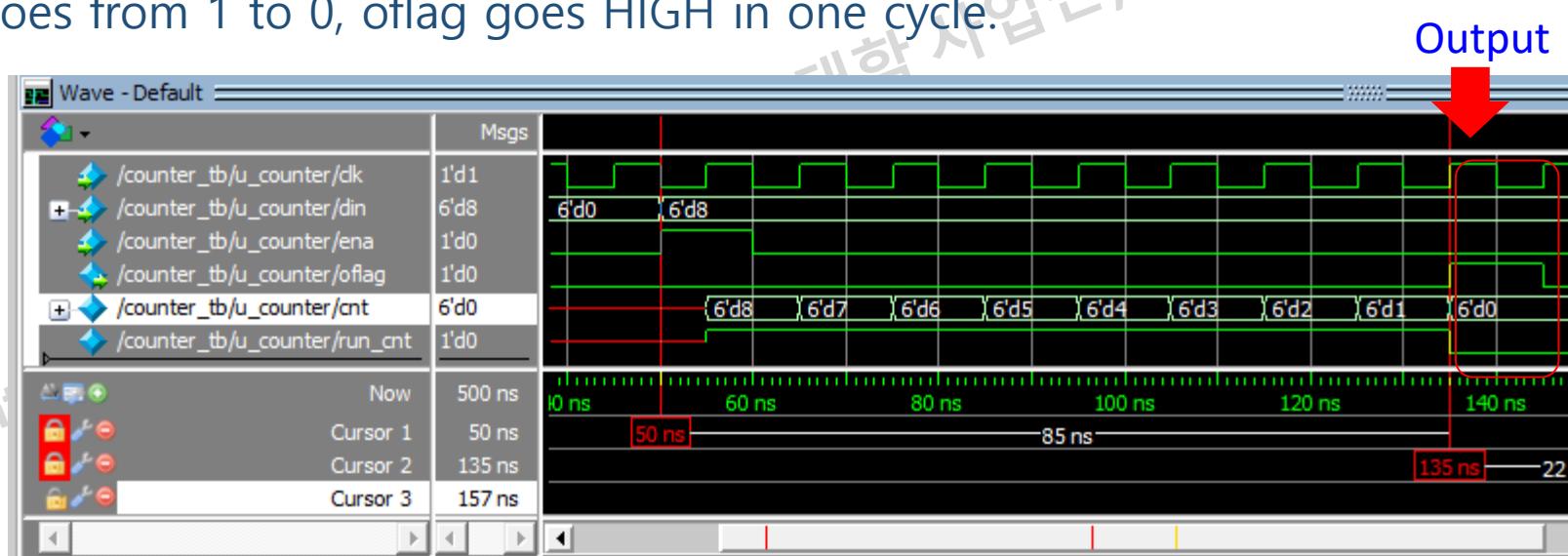
- Start
 - 'ena' goes HIGH and keeps in one cycle
 - A new value is written to 'din'
- Counting
 - At rising (positive) edge of 'clk', internal 'cnt' deceases by 1 if possible
- Output
 - 'cnt' goes from 1 to 0, oflag goes HIGH in one cycle.

Counting



Waveform

- Start
 - 'ena' goes HIGH and keeps in one cycle
 - A new value is written to 'din'
- Counting
 - At rising (positive) edge of 'clk', internal 'cnt' deceases by 1 if possible
- Output
 - 'cnt' goes from 1 to 0, oflag goes HIGH in one cycle.



To do ...

- Implement counter.v and counter_tb.v by completing the missing codes
- Do a simulation with time = 500ns
- Show the output waveform

```
'timescale 1ns/1ps

`define CLOCK_PERIOD 10 //100MHz
module counter_tb;

reg clk;
reg [5:0] din;
reg ena;
wire oflag;

// Counter design under test
counter
u_counter(.PORTNAME(clk),
           .PORTNAME(din),
           .PORTNAME(ena),
           .PORTNAME(oflag));
```

```
module counter(clk, din, ena, oflag);
input clk;
input [5:0] din;
input ena;
output reg oflag;
reg [5:0] cnt;

wire run_cnt;
//-----
// Internal counter: cnt
//-----
assign run_cnt = (cnt!=0)?1'b1: 1'b0; // Check if cnt is zero
                                         // 1: Not zero, 0: zero

always@(posedge clk) begin
    if(run_cnt)begin
        // Insert your code here
    end
    else if (ena) begin
        // Insert your code here
    end
end

//-----
// Output: oflag
//-----
always@(posedge clk) begin
    if(cnt == 1) begin
        // Insert your code here
    end
    else begin
        // Insert your code here
    end
end

endmodule
```

Copyright

Questions

1. What is the clock frequency used in Lab3?
 - Change the clock frequency to 50MHz
 - Do simulation
 - Capture the new waveform.
2. To update the internal counter, can we change the logic order (a) to (b)?
 - Show a test case and its waveform to distinguish the logics (a) and (b).

```
always@(posedge clk) begin
    if(run_cnt)begin
        // Insert your code here
    end
    else if (ena) begin
        // Insert your code here
    end
end
```

(a)

```
always@(posedge clk) begin
    if(ena)begin
        // Insert your code here
    end
    else if (run_cnt) begin
        // Insert your code here
    end
end
```

(b)