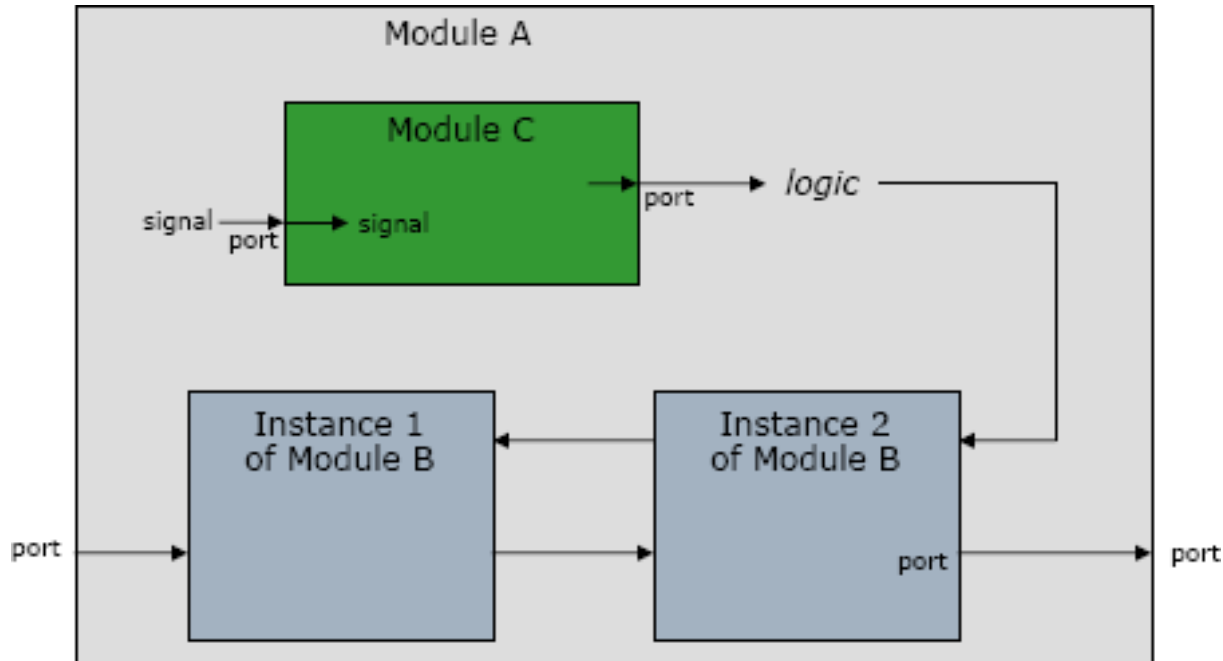


# Computer Architecture Lab

Lab03 – Week #3

# Instantiation

- Design inside designs
- Higher-level designs instantiate lower-level designs
- Applies to both schematic designs and HDL designs



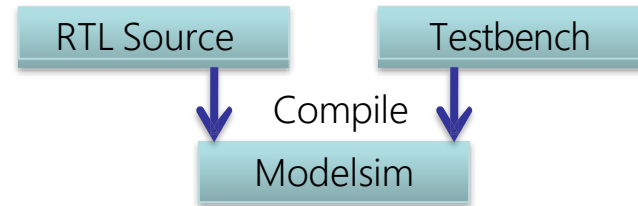
# Simulation types

---

- Function Simulation
  - ✓ Simulate with the pure RTL code and testbench that you wrote
  - ✓ Simulation results don't include time delay
  
- Gate (level) Simulation
  - ✓ Simulate with simulation files that generated after synthesizing or implementing RTL code
  - ✓ After RTL code synthesized, simulation lib is required
  - ✓ When simulating, a consistent delay value is allocated per gate
  
- Timing Simulation
  - ✓ It is a simulation that proceeds after implementation, and uses the sdf file
  - ✓ Need lib for simulation
  - ✓ Simulation applies to all delay of Gate, Net, and PAD, and the simulation time is extended due to the large number of operations, and results in almost the same behavior as the actual FPGA

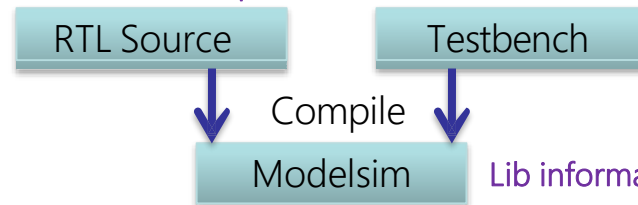
# Simulation's conceptual diagram

## ➤ Function Simulation



## ➤ Gate(Level) Simulation

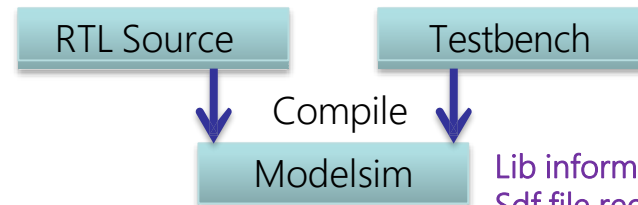
HDL source created after  
Post Synthesis or Translate operation



Lib information required

## ➤ Timing Simulation

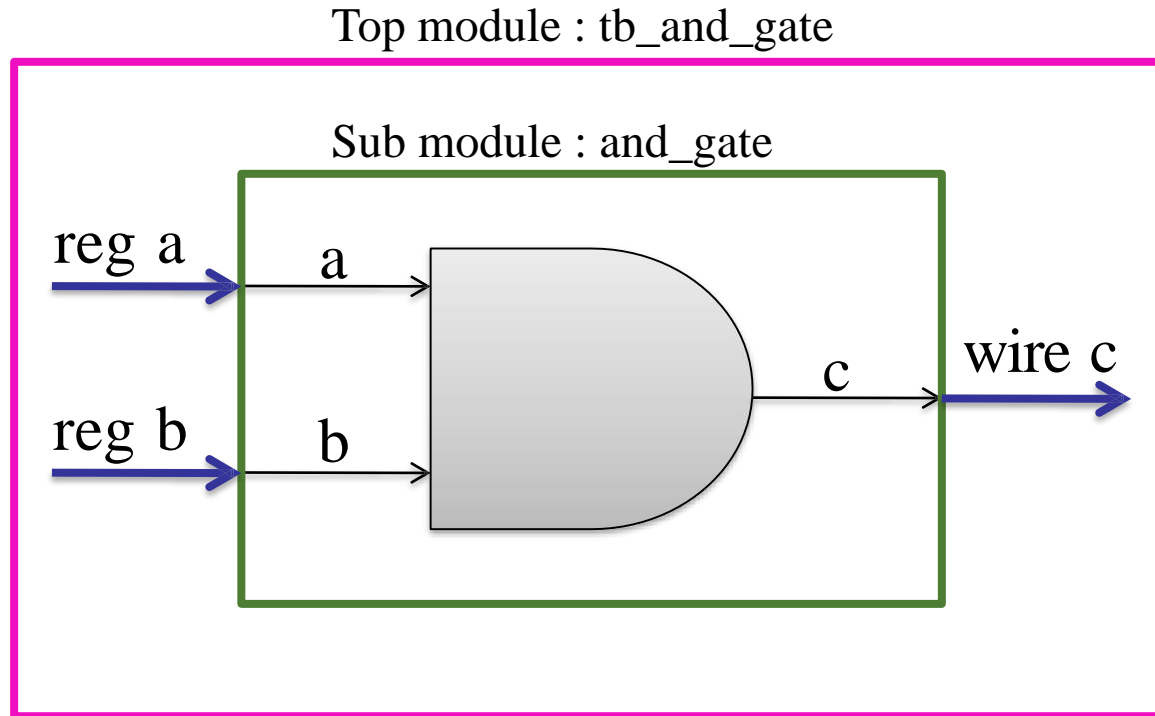
HDL source and sdf file created  
after Place & Route operation



Lib information required  
Sdf file required

# Relationship between testbench and module

---



# Testbench (1/4)

---

## ➤ Definition

- ✓ Module for simulation
- ✓ Used to validate a designed module
- ✓ Contains the designed module, generates the input value of the module and checks the output value

<testbench>

input signal ➡ Module to validate ➡ output signal

# Testbench (2/4)

---

## ➤ System function

- ✓ Only used on testbench
- ✓ "\$" start with a symbol
- ✓ \$stop – Simulation stop
- ✓ \$finish – End of simulation
- ✓ \$time – got current simulation time
- ✓ \$monitor, \$display – used to display specific values

```
initial begin  
#300;  $finish;  
end
```

# Testbench (3/4)

---

## ➤ ``timescale` (Not `'`)

- ✓ ``timescale <testbench step unit>/ <simulation step unit>`

  - ex) ``timescale 1ns/1ps`

- ✓ Specify the unit of time at the top of the testbench

- ✓ Do not add a semicolon to the end of a sentence

- ✓ The testbench step unit is a unit of time delay, etc.

  - ex) If declared ``timescale 1ns/1ps`, `#30` means 30ns

- ✓ Determine the resolution of the time shown in the timing diagram in units of simulation steps



# Testbench (4/4)

---

- “initial” statement
  - ✓ “initial” block is a block that runs only once when simulation starts
  - ✓ Available only on testbench
- Time delay
  - ✓ You can use the “initial” statement to generate the input value of the module

```
initial begin
    // X would be not changed for 30[ns]
    X = 1; #30;
end
```

# Testbench example

```
`timescale 1ns/1ns                                //step time
module tb_and_gate;                                //module name
    parameter STEP = 100;                          //clock period
    reg tb_a, tb_b;                                //declare input as a reg
    wire tb_c;                                       //declare output as a wire
    //call module
    and_gate TEST(.a(tb_a), .b(tb_b), .c(tb_c));

    initial
    // begin ~ end is sequential block
    begin
        tb_a = 0;  tb_b = 0;
        #STEP;     tb_a = 0;      tb_b = 0;
        #STEP;     tb_a = 0;      tb_b = 1;
        #STEP;     tb_a = 1;      tb_b = 0;
        #STEP;     tb_a = 1;      tb_b = 1;
        #STEP;     $stop;
        //interrupt the simulation and change to conversation mode
        #STEP;     $finish; // $finish : terminate the simulation
    end
endmodule
```

# Logic gate Review (1/3)

---

## ➤ Combinational Logic

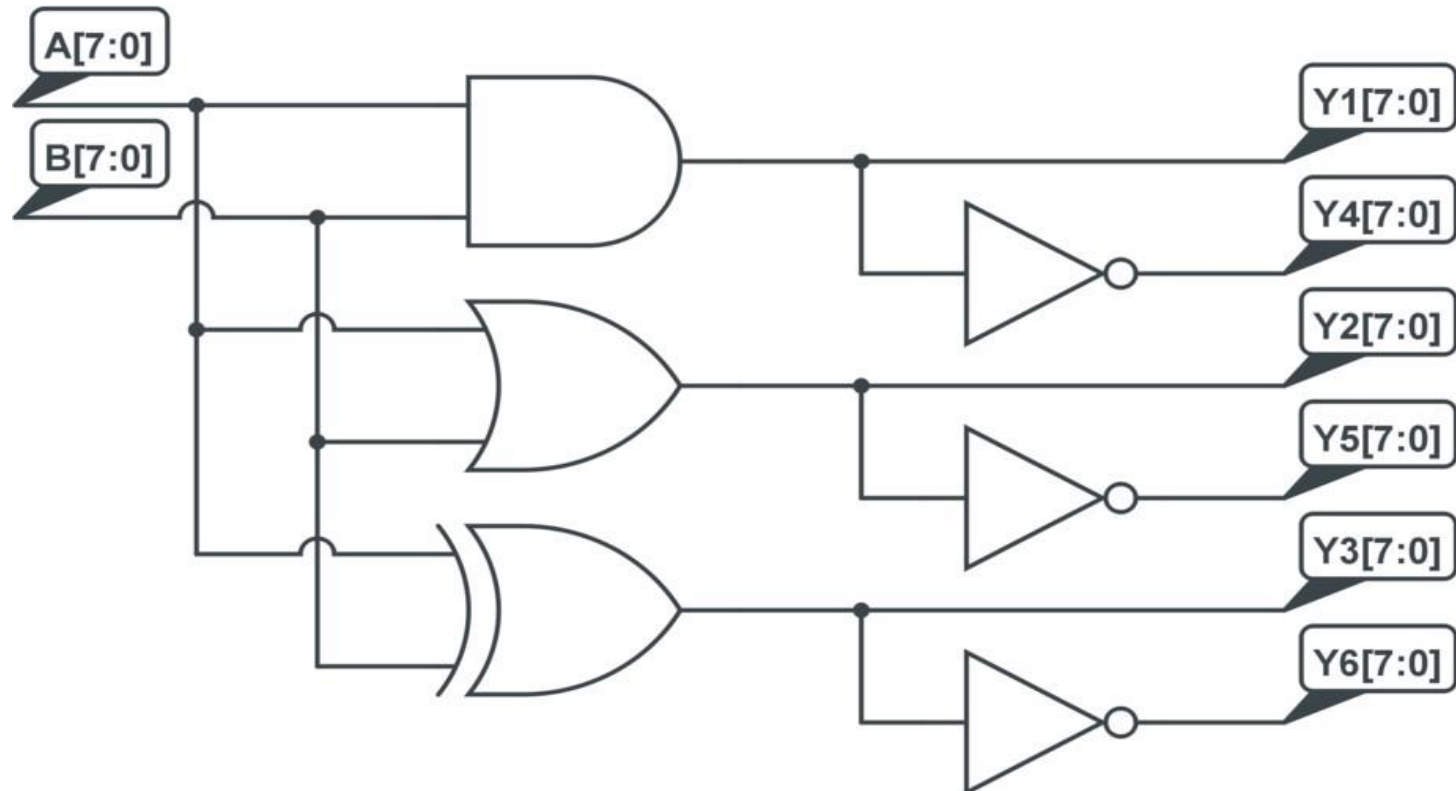
- ✓ Digital logic type that output is determined only by the current input

## ➤ Bitwise Operator

Operator	Function
	OR
&	AND
~	NOT
^	XOR

# Logic gate Review (2/3)

## ➤ 8-bit Logic Gates



# Logic gate Review (3/3)

---

## ➤ Verilog HDL code

### ✓ Top module

```
module gates(a, b, y1, y2, y3, y4, y5, y6);
```

```
input [7:0] a, b;
```

```
output [7:0] y1, y2, y3, y4, y5, y6;
```

```
assign y1 = a & b;
```

```
assign y2 = a | b;
```

```
assign y3 = a ^ b;
```

```
assign y4 = ~(a & b);
```

```
assign y5 = ~(a | b);
```

```
assign y6 = ~(a ^ b);
```

```
// assign out = in1 op in2 is called continuous assignment.
```

```
// The right side of the = means input, and the left side means output.
```

```
endmodule
```

# Combinational logic

---

- Example of "Always" Statement with Combinational logic

```
wire [3:0] DEC;
reg [3:0] Y;

always@ ( DEC )
begin
    case(DEC)
        4'h1 : Y = 4'b0001;
        4'h2 : Y = 4'b0010;
        4'h4 : Y = 4'b0100;
        4'h8 : Y = 4'b0000;
        default : Y = 4'b0000;
    endcase
end
```

# Sequential logic (1/4)

---

- “always” statement within sequential logic
  - ✓ You should use “always” statements when you construct the sequential logic
  - ✓ Declare signal as a “reg” in the “always” statement
  - ✓ Use only “clock” and “reset” for eventlist in “always” statements
  - ✓ Use non-blocking(“<=”) in the sequential logic
  - ✓ Non-blocking(“<=”) is to perform multiple sentences of begin-end block at the same time
  - ✓ The “reset” is to use negative edge (negedge) as much as possible
  - ✓ The “reg” used in “always” statements is flip-flop (reset is required)
  - ✓ You must start the “reset” operation first in the “always” statement
  - ✓ Multiple “always” statements are available in one module
    - (Do not assign the same register with different “always” statements)

## Sequential logic (2/4)

---

```
always @ (posedge clock or negedge reset)
begin
  if(!reset)      //reset started, low active
    Y<=0;        // use "<="
  else begin
    if(A>B)
      Y <= A;
    else
      Y <= B;
  end
end
end
```



# Sequential logic (3/4)

---

- Caution of using always statement within sequential logic (1)
  - ✓ You can't use vector index in the edge of the clock  
ex) always @(posedge CLK[1]) // error
  - ✓ You can't use vector bit in the reset condition  
ex) always @(posedge CLK or negedge RESET\_BUS) // error
    - if(!RESET\_BUS[1]) // error
  - ✓ You can't use complex functions in the reset condition  
ex) always @(posedge CLK or negedge RESET) if(RESET == (1-1))  
// error

# Sequential logic (4/4)

---

- Caution of using “always” statement within sequential logic (2)
  - ✓ You can only use RESET signal in the reset condition
    - ex) always @(posedge CLK or negedge RESET)
    - if(!RESET||zero\_init) // error
  - ✓ When you use “if” statement in the “always” statement, you should use “if” statement first
- ex) always @(posedge CLK or negedge RESET)
  - A<=3; // error
  - if(!RESET)

# Sequential logic & Combinational logic

---

## ➤ Syntax

- ✓ Combinational logic : assign statement, "always" statement,

- Use " = "

- ✓ Sequential logic : "always" statement,

- Use " <= "

## ➤ Data type

- ✓ assign statement : wire declaration

- ✓ always statement : reg declaration

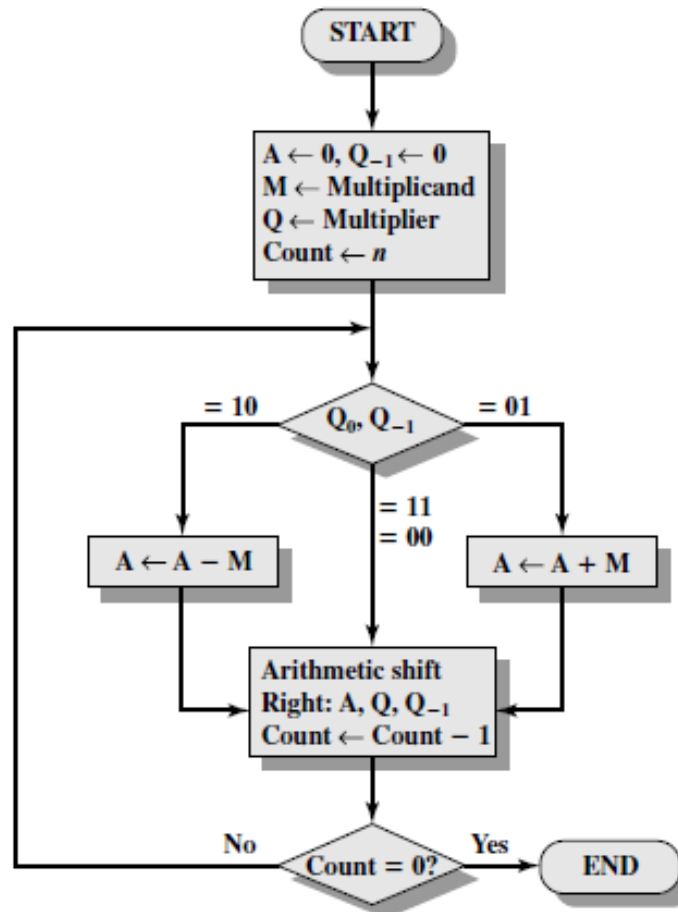
# Blocking vs non-blocking

## ➤ Operator

- ✓ Blocking(=)
- ✓ Non-Blocking(<=) : Only using in "always" statement
  - Example : Use in Shift register

	Blocking	Non-Blocking
code	wire clk, A reg A, B, C always@(posedge clk) b egin B = A; C = B; End	wire clk, A r eg A, B, C always@(posedge clk) b egin B <= A; C <= B; end
Initialized value	A = 1, B = 2, C = 3	A = 1, B = 2 , C = 3
Result	A = 1, B = 1, C = 1	A = 1, B = 1, C = 2

# Booth algorithm diagram



# Booth algorithm (1/9)

➤ ex)  $-5 \times -4 = +20$

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

M	A	Q	Q-1
1011	0000	1100	0

Shift right

# Booth algorithm (2/9)

➤ ex)  $-5 \times -4 = +20$

✓ iterate = 1

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0

↑  
Arithmetic  
Shift right

# Booth algorithm (3/9)

➤ ex)  $-5 \times -4 = +20$

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0

Shift right



# Booth algorithm (4/9)

➤ ex)  $-5 \times -4 = +20$

✓ iterate = 2

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0
1011	0000	0011	0

Arithmetic  
Shift right

# Booth algorithm (5/9)

➤ ex)  $-5 \times -4 = +20$

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0
1011	0000	0011	0

Subtract and Shift right

# Booth algorithm (6/9)

➤ ex)  $-5 \times -4 = +20$

✓ iterate = 3

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

$\bar{M} = 0100 + 1 = 0101$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0
1011	0000	0011	0
1011	0101	0011	0

Subtract  
result

# Booth algorithm (7/9)

➤ ex)  $-5 \times -4 = +20$


✓ iterate = 3

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

$\bar{M} = 0100 + 1 = 0101$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0
1011	0000	0011	0
1011	0101	0011	0
1011	0010	1001	1


  
 Arithmetic  
Shift right

# Booth algorithm (8/9)

➤ ex)  $-5 \times -4 = +20$

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

$\bar{M} = 0100 + 1 = 0101$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0
1011	0000	0011	0
1011	0101	0011	0
1011	0010	1001	1

Shift right

# Booth algorithm (9/9)

➤ ex)  $-5 \times -4 = +20$

✓ iterate = 4

$M(\text{multiplicand}) = 1011 (-5)$

$Q(\text{multiplier}) = 1100 (-4)$

$\bar{M} = 0100 + 1 = 0101$

M	A	Q	Q-1
1011	0000	1100	0
1011	0000	0110	0
1011	0000	0011	0
1011	0101	0011	0
1011	0010	1001	1
1011	0001	0100	1
$2^4 + 2^2 = 16 + 4 = 20$			

# Booth verilog code (1/2)

## ➤ Booth.v

```
module booth(input1, input2, clk, start, reset, result, count);

    input [3:0] input1, input2;
    input clk, start, reset;
    output [7:0] result;
    output reg [1:0] count;

    reg [3:0] A, Q, M;
    reg Q_1;
    wire [3:0] add, sub;

    always @(posedge clk or negedge reset) begin
        if(~reset)
            begin //reset
                A <= 4'b0;
                M <= 4'b0;
                Q <= 4'b0;
                Q_1 <= 1'b0;
                count <= 2'b0;
            end

        else if(~start)
            begin //start
                A <= 4'b0;
                M <= input1;
                Q <= input2;
                Q_1 <= 1'b0;
                count <= 2'b0;
            end

        else
            begin
                case({Q[0],Q_1})
                    2'b0_1 : {A, Q, Q_1} <= {add[3], add, Q}; //A + M and shift right
                    2'b1_0 : {A, Q, Q_1} <= {sub[3], sub, Q}; //A + ~M and shift right
                    default : {A, Q, Q_1} <= {A[3], A, Q}; //shift right
                endcase

                count = count + 1'b1;
            end
        end

        alu adder(A, M, 1'b0, add);
        alu subtracter(A, ~M, 1'b1, sub);

        assign result = {A,Q};
    endmodule
```

```
module alu(a, b, cin, out);
    input [3:0] a;
    input [3:0] b;
    input cin;
    output [3:0] out;

    assign out = a + b + cin;
endmodule
```

# Booth verilog code (2/2)

## ➤ tb\_Booth.v

```
`timescale 1ns/1ps
`include "booth.v"
module tb_booth;
    reg [3:0] input1, input2;
    reg clk, start, reset;
    wire [7:0] result;
    wire [1:0] count;

    booth Booth(.input1(input1), .input2(input2), .clk(clk), .start(start), .reset(reset), .result(result), .count(count));

    always #5 clk = ~clk;

    initial begin
        $dumpfile("tb_booth.vcd");
        $dumpvars(0,tb_booth);

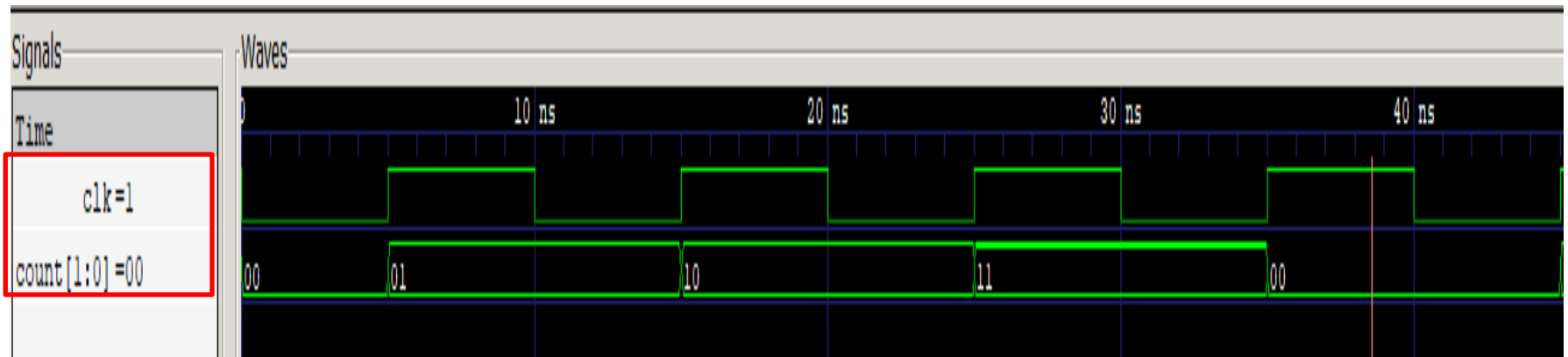
        reset = 1;
        start = 1;
        clk = 0;
        reset = 0;
        #10;

        start = 0;
        reset = 1;
        input1 = 4;
        input2 = 5;
        #10;
        start = 1;
        #640; $finish;
    end
endmodule
```

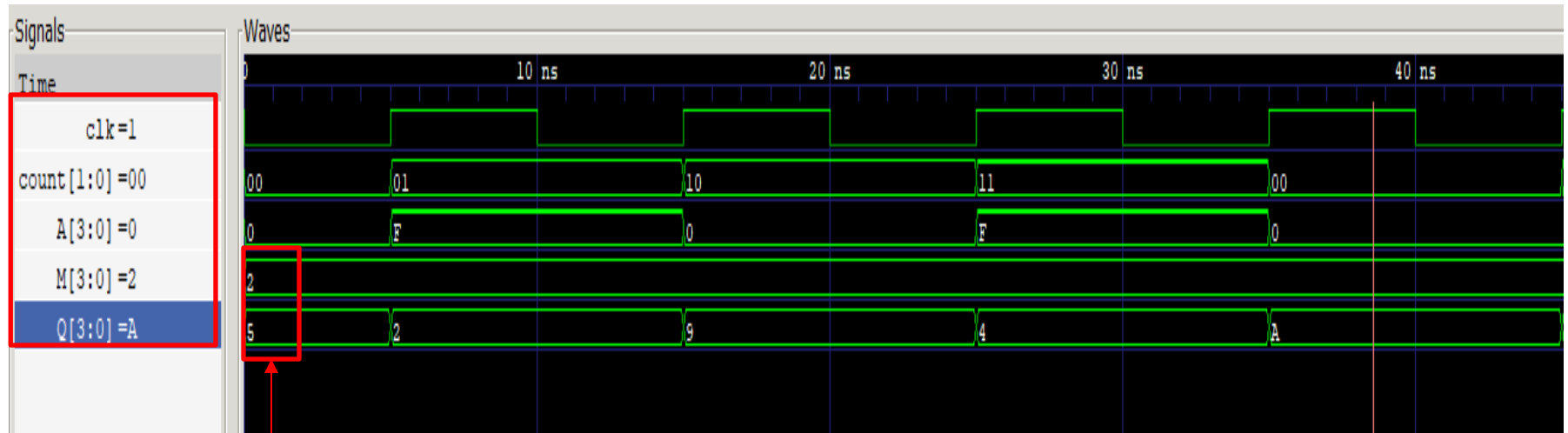


# Booth waveform (1/4)

---

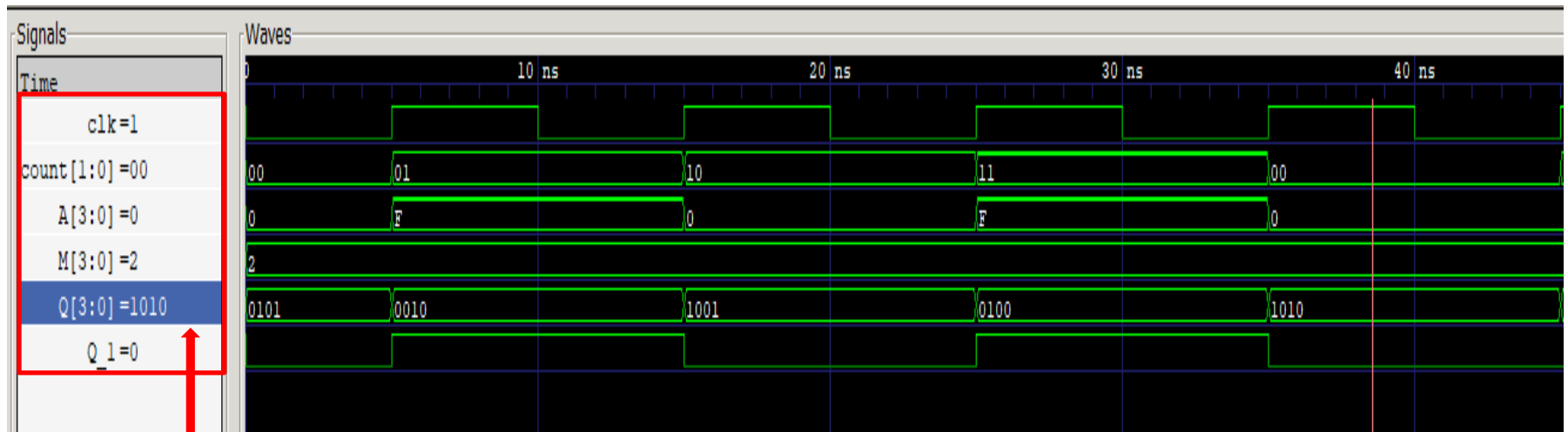


# Booth waveform (2/4)



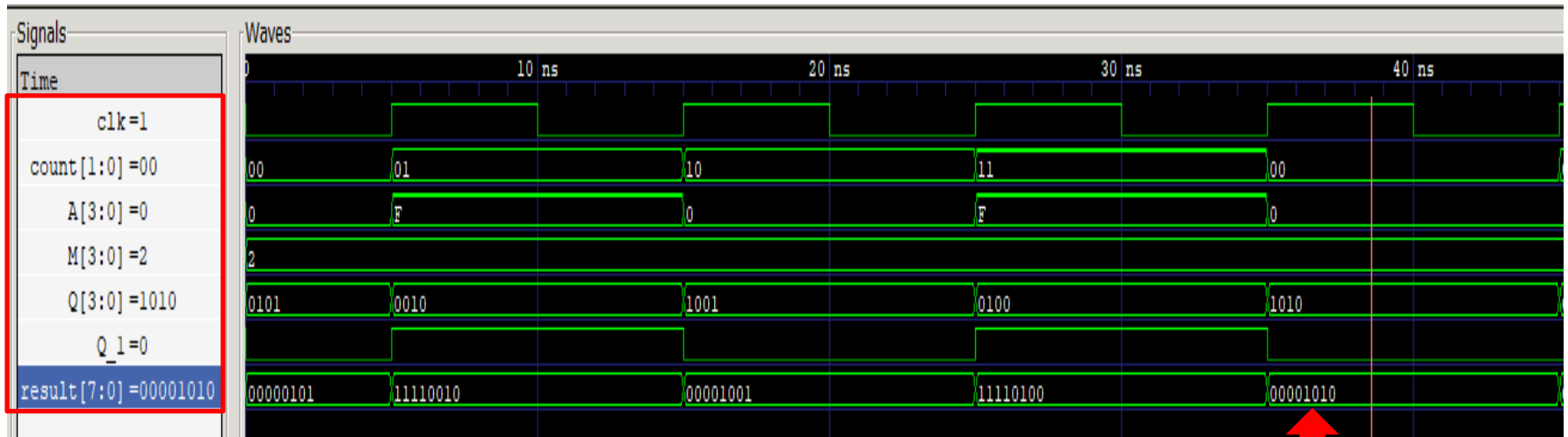
inputs

# Booth waveform (3/4)



Change  
to binary

# Booth waveform (4/4)



$$2^3 + 2 = 10$$

**Thank You**