Lab # 08

Verilog Behavioral Modeling (combinational Circuits).

Objectives:

- To construct combinational digital circuits (e.g., adders, subtractors, multiplexers, decoders) using behavioral modeling in Verilog HDL.
- To apply simulation tools (Vivado) for the design, implementation, and verification of behavioral Verilog codes.
- To develop behavioral models of combinational logic using procedural blocks like always and initial.
- To differentiate and implement blocking (=) and non-blocking (<=) assignments appropriately within behavioral descriptions.
- To simulate and analyze circuit functionality through test benches and waveform inspection using modern tools.
- To construct efficient Verilog designs that demonstrate the relationship between high-level behavioral code and underlying hardware logic.

Introduction

Verilog provides designers with the ability to describe design functionality algorithmically. In other words, the designer describes the **circuit's behavior**. Thus, behavioral modeling represents the circuit at a very high level of abstraction. Design at this level resembles C programming more than it resembles digital circuit design. Behavioral Verilog constructs are like C language constructs in many ways. Verilog is rich in behavioral constructs that provide the designer with a great amount of flexibility.

Structured Procedures

There are two structured procedure statements in Verilog: *always* and *initial*. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

Verilog is a concurrent programming language, unlike the C programming language, which is sequential in nature. Activity flows in Verilog run in parallel rather than in sequence. Each always and initial statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements always and initial cannot be nested. The fundamental difference between the two statements is explained in the following sections.

a) Initial Statement

All statements inside an initial statement constitute an **initial block**. An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks. Multiple behavioral statements must be grouped; typically, using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary. This is similar to the { } grouping in the C programming language.

Example:

```
initial
m = 1'b0; //single statement; does not need to be grouped
initial begin
#5 a = 1'b1; //multiple statements; need to be grouped
#25 b = 1'b0;
end
initial begin
#10 x = 1'b0;
#25 y = 1'b1;

end
initial
#50 $finish;
```

In the above example, the three initial statements start to execute in parallel at time 0. If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the initial blocks will be as follows.

```
time    statement executed 0 m = 1'b0;
5       a = 1'b1;
10      x = 1'b0;
30      b = 1'b0;
35      y = 1'b1;
50      $finish;
```

The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

b) Always Statement

All behavioral statements inside an always statement constitute an **always block**. The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. An example is a clock generator module that toggles the clock signal every half cycle. In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on.

Example:

```
//Initialize clock at time zero
initial
clock = 1'b0;

//Toggle clock every half-cycle
always
#10 clock = ~clock;
initial
#1000
 $finish;
```

In the example, the always statement starts at time 0 and executes the statement clock = ~clock every 10 time units. Notice that the initialization of the clock must be done inside a separate initial statement. If we put the initialization of the clock inside the always block, the clock will be initialized every time the always block is entered. Also, the simulation must be halted inside an initial statement. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever.

C programmers might distinguish between the always block and an infinite loop. But hardware designers tend to view it as a continuously repeated activity in a digital circuit, starting from power on. The activity is stopped only by a power off ($finish) or by an interrupt ($stop).

Procedural Assignment

Procedural assignments update reg, integer, real, or time variable values. a different value. These are unlike continuous assignments discussed in Dataflow Modeling, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net.

Timing Controls

Various behavioral timing control constructs are available in Verilog. In Verilog, if there are no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at which procedural statements will execute. There are three methods of timing control:

• delay-based timing control
• event-based timing control
• level-sensitive timing control

Delay-Based Timing Control

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. There are three types of delay control for procedural assignments:

- • regular delay control
- • intra-assignment delay control
- • zero delay control

Regular event control

The @ symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value. The keyword posedge is used for a positive transition, as shown in the example.

Example

@(clock) q = d; //**q = d is executed whenever signal clock changes value**

@(posedge clock) q = d; //**q = d is executed whenever signal clock does a positive transition**

Event OR Control

Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals. The list of events or signals expressed as an OR is also known as a sensitivity list. The keyword or is used to specify multiple triggers, as shown in the Example.

Example

```
always @ (reset or clock or d) //Wait for reset or clock or d to change
begin
if (reset) //if reset signal is high, set q to 0
q = 1'b0;
else if(clock) //if clock is high, q = d
q = d;
end
```

Sensitivity lists can also be specified using the "," (comma) operator instead of the or operator. Comma operators can also be applied to sensitivity lists that have edge-sensitive triggers. The above example can be rewritten using the comma operator, the only change would be:

```
always @( reset, clock, d)
```

When the number of input variables to a combination logic block are very large, sensitivity lists can become very cumbersome to write. To solve this problem, Verilog HDL contains two special symbols: @* and @(*). Both symbols exhibit identical behavior. These special symbols are sensitive to a change on any signal that may be read by the statement group that follows this symbol.

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e)
begin
out1 = a ? b+c : d+e;
end
//Instead of the above method, use @(*) symbol. Alternately, the @* symbol can be used
//All input variables are automatically included in the sensitivity list
always @(*)
begin
out1 = a ? b+c : d+e;
end
```

Level-Sensitive Timing Control

Event control discussed earlier waited for the change of a signal value or the triggering of an event. The symbol @ provided edge-sensitive control. Verilog also allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed. The keyword wait is used for level-sensitive constructs.

Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executed. Keywords if and else are used for conditional statements. There are three types of conditional statements. Usage of conditional statements is shown below.

```
//Type 1 conditional statement. No else statement
//Statement executes or does not execute

if (<expression>) true_statement ;
//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated

if (<expression>) true_statement ; else false_statement ;
//Type 3 conditional statement. Nested if-else-if
//Choice of multiple statements. Only one is executed.

if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

The <*expression*> is evaluated. If it is true (1 or a non-zero value), the *true_statement* is executed. However, if it is false (zero) or ambiguous (x), the *false_statement* is executed. The <*expression*> can contain any operators used in dataflow modeling. Each *true_statement* or *false_statement* can be a single statement or a block of multiple statements. A block must be grouped, typically by using keywords **begin** and **end**. A single statement need not be grouped.

Multiway Branching

In the type 3 conditional statement in the previous section, there were many alternatives, from which one was chosen. The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.

The keywords case, endcase, and default are used in the case statement.

case (expression)

alternative1: statement1;

alternative2: statement2;

alternative3: statement3;

...

...

default: default_statement;

endcase

Each of *statement1, statement2 ..., default_statement* can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords **begin** and **end**. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives match, the *default_statement* is executed. The *default_statement* is optional. Placing multiple default statements in one case statement is not allowed. The case statements can be nested. The case statement can also act like a many-to-one multiplexer

Example: 4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3; input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals.
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
2'd3 : out = i3;
default: $display("Invalid control signals");
endcase
endmodule
```

The case statement compares 0, 1, x, and z values in the expression and the alternative bit for bit. If the expression and the alternative are of unequal bit width, they are zero filled to match the bit width of the widest of the expression and the alternative.

Example and gate (Behavioral modelling)

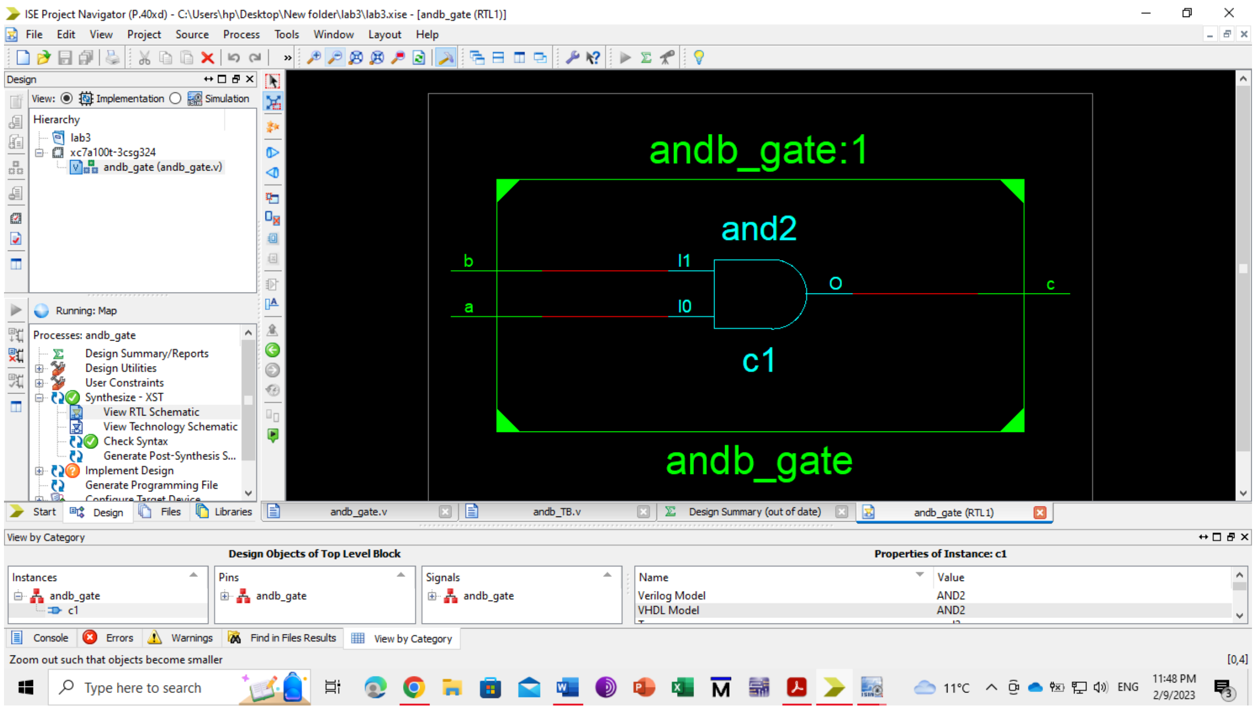
Code

```
module andb_gate(c,a,b);
input a,b;
output c;
reg c;
always@(a or b)
c = a & b;
endmodule
```

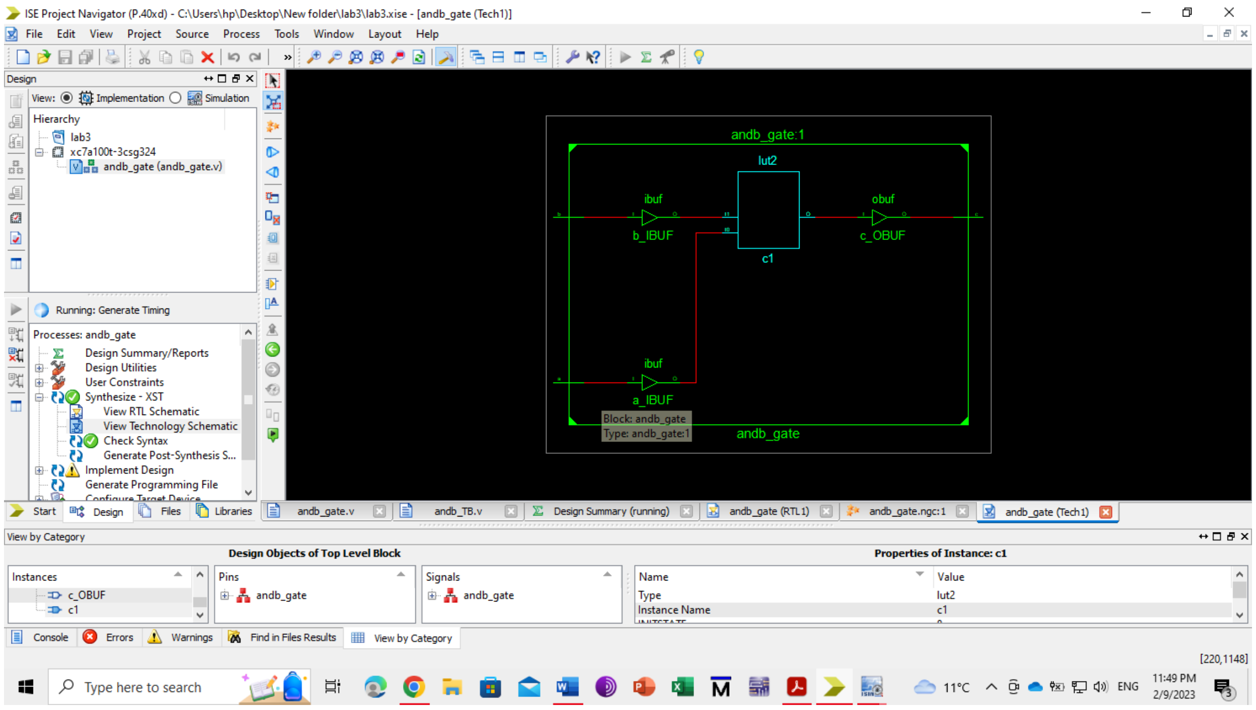
Test Bench

```
module andb_TB;
        reg a;
        reg b;
        wire c;
        andb_gate andg(c,a,b);
        initial
        begin
        //$monitor($time,"c=%b,a=%b,b=%b",c,a,b);
                a = 0; b = 0;
                #20  a= 0; b = 1;
                #20  a= 1; b = 0;
                #20  a= 1; b = 1;
                #20 $finish;
        end

endmodule
```

RTL

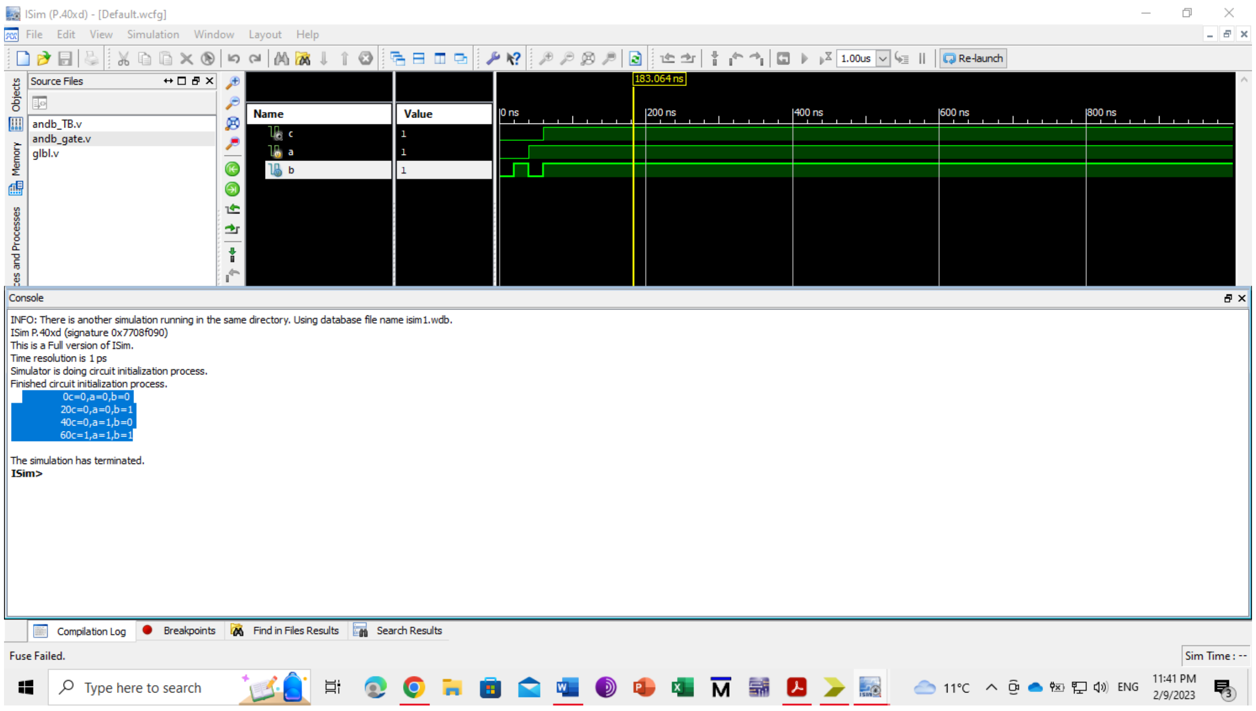

ISE Project Navigator (P.40xd) - C:\Users\hp\Desktop\New folder\lab3\lab3.xise - [andb_gate (RTL1)]
andb_gate:1
and2
b
a
I1
I0
O
c
c1
andb_gate
Running: Map
Processes: andb_gate
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
View RTL Schematic
View Technology Schematic
Check Syntax
Implement Design
Generate Programming File
Design Objects of Top Level Block
Properties of Instance: c1
Zoom out such that objects become smaller

Name	Value
Verilog Model	AND2
VHDL Model	AND2


ISE Project Navigator (P.40xd) - C:\Users\hp\Desktop\New folder\lab3\lab3.xise - [andb_gate (Tech1)]
andb_gate:1
lut2
ibuf
b_IBUF
obuf
c_OBUF
c1
ibuf
a_IBUF
Block: andb_gate
Type: andb_gate:1
andb_gate
Running: Generate Timing
Design Objects of Top Level Block
Properties of Instance: c1

Name	Value
Type	lut2
Instance Name	c1

Waveform



```
INFO: There is another simulation running in the same directory. Using database file name isim1.wdb.
ISim P.40xd (signature 0x7708f090)
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
                   0c=0,a=0,b=0
                  20c=0,a=0,b=1
                  40c=0,a=1,b=0
                  60c=1,a=1,b=1

The simulation has terminated.
ISim>
```

Example # 2 Half Adder (Behavioral modelling)

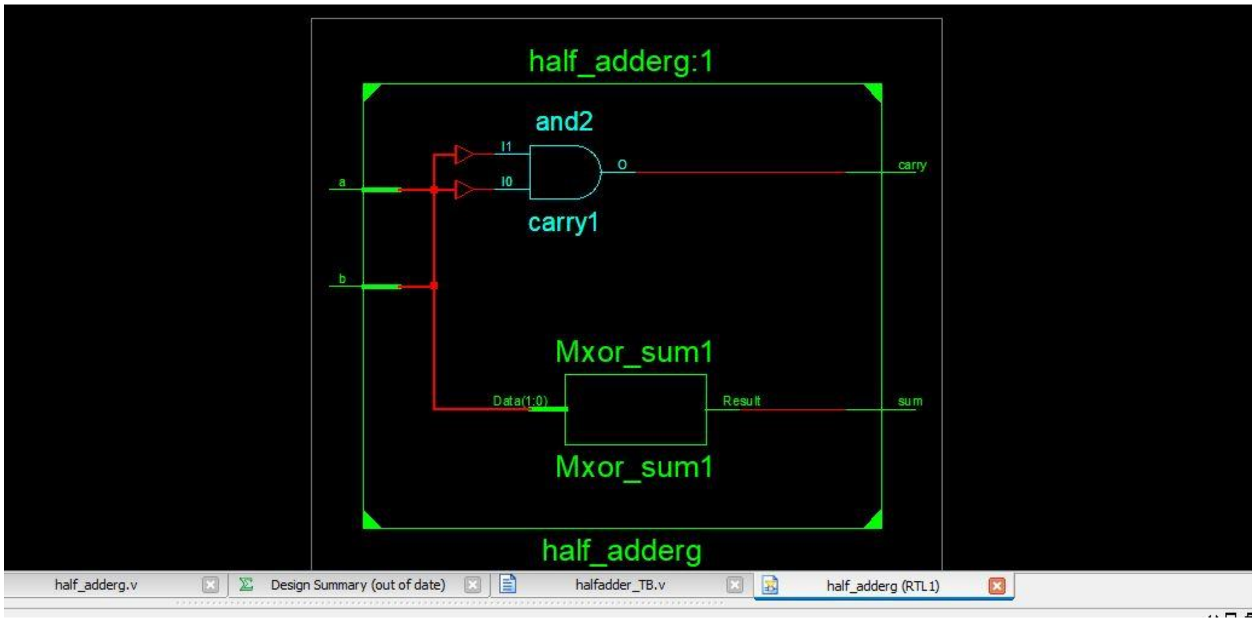
Code

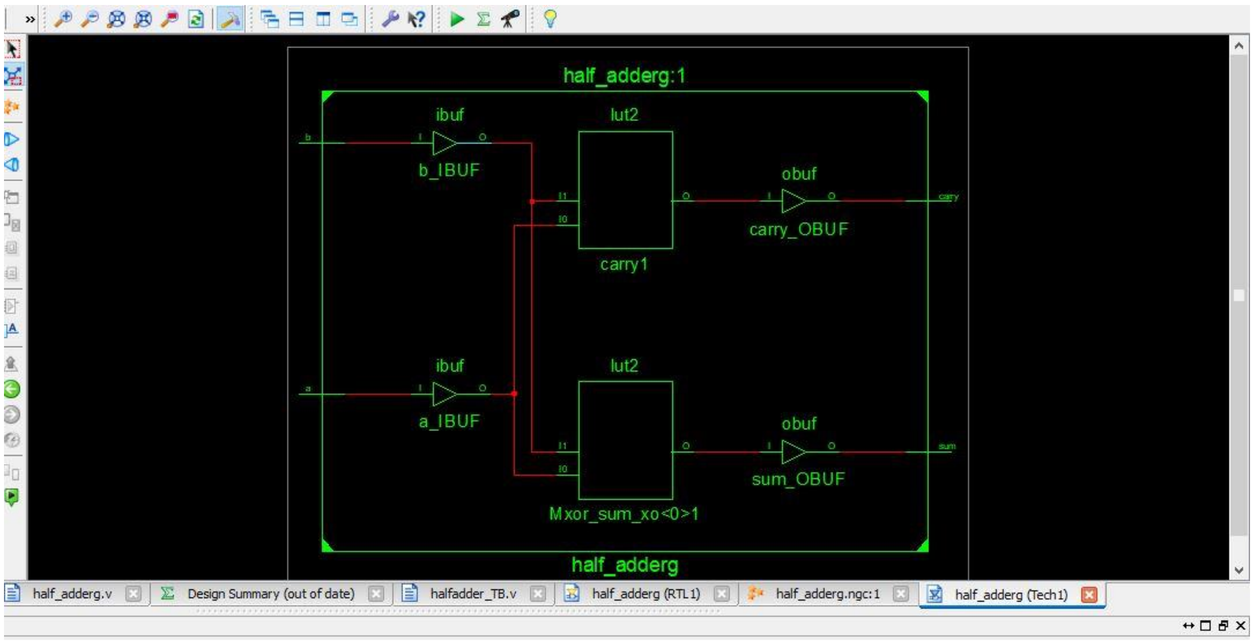
```
module half_adderg(sum,carry,a,b);
input a,b;
output sum,carry;
reg sum,carry;
// always @ (*)
always@(a or b)
begin
sum = a^b;
carry = a&b;
end
endmodule
```

Test Bench

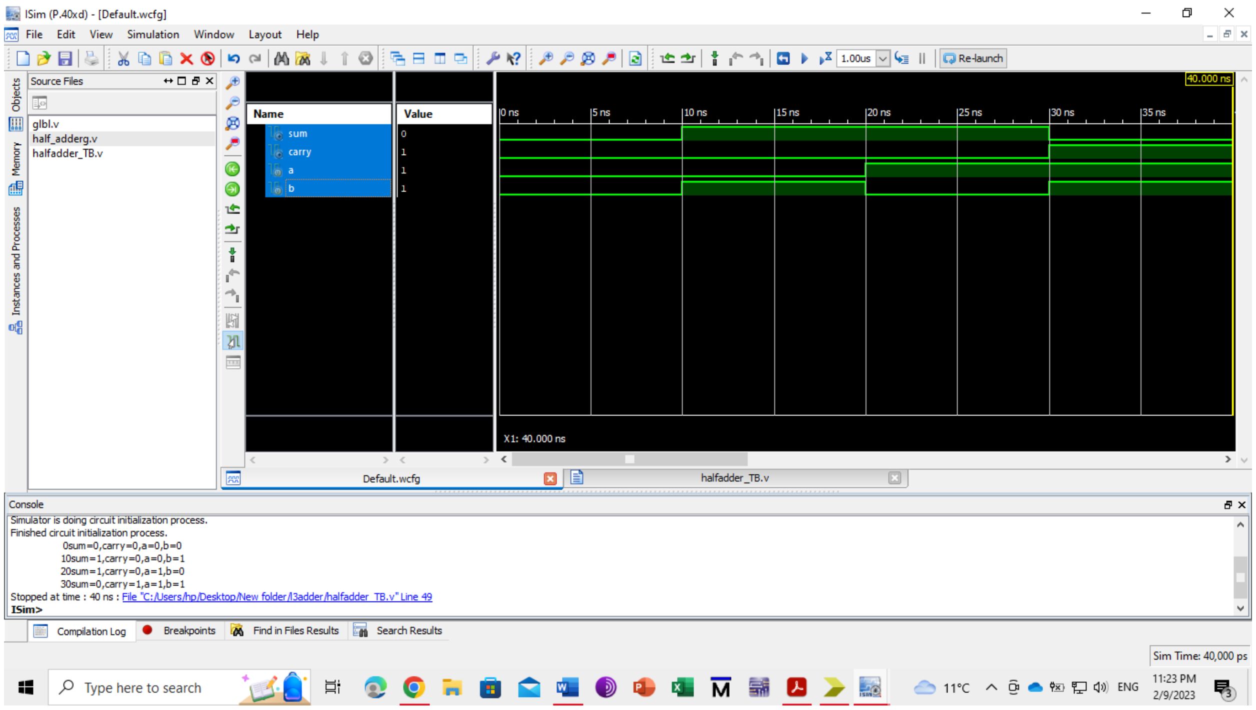
```
module halfadder_TB;
        // Inputs
        reg a;
        reg b;
        // Outputs
        wire sum;
        wire carry;
        half_adderg gate(sum,carry,a,b);
        initial
        begin
//$monitor($time, "sum=%b,carry=%b,a=%b,b=%b",sum,carry,a,b);

                a = 0; b = 0;

#10     a = 0; b = 1;

#10     a = 1; b = 0;

#10     a = 1; b = 1;

#10 $finish;
  end
endmodule
```

RTL


half_adderg:1
and2
I1
I0
O
carry1
a
b
carry
Mxor_sum1
Data(1:0)
Result
sum
Mxor_sum1
half_adderg
half_adderg.v
Design Summary (out of date)
halfadder_TB.v
half_adderg (RTL1)


half_adderg:1
ibuf
b_IBUF
lut2
carry1
obuf
carry_OBUF
carry
ibuf
a_IBUF
lut2
Mxor_sum_xo<0>1
obuf
sum_OBUF
sum
half_adderg
half_adderg.v
Design Summary (out of date)
halfadder_TB.v
half_adderg (RTL1)
half_adderg.ngc:1
half_adderg (Tech1)

Waveform



```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
          0sum=0,carry=0,a=0,b=0
         10sum=1,carry=0,a=0,b=1
         20sum=1,carry=0,a=1,b=0
         30sum=0,carry=1,a=1,b=1
Stopped at time : 40 ns : File "C:/Users/hp/Desktop/New folder/l3adder/halfadder_TB.v" Line 49
ISim>
```

Example # 3 {8x1} multiplexer (Behavioral modelling)

Multiway branching (case statement)

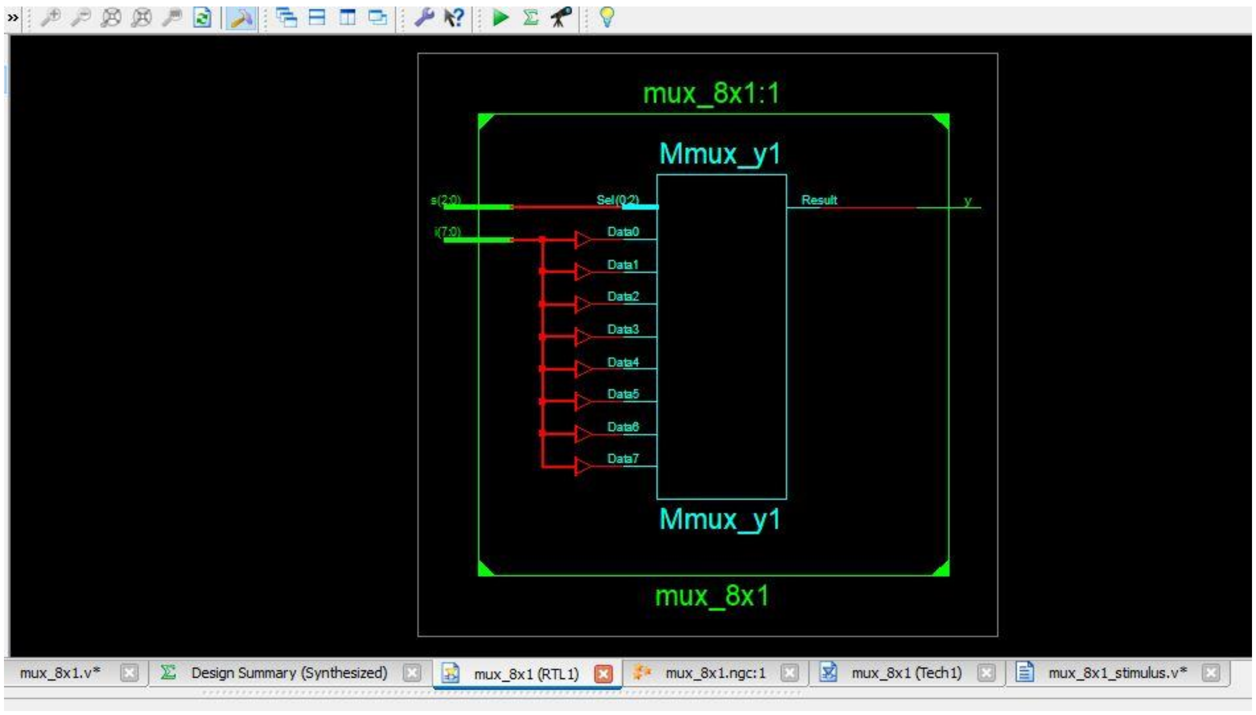
Code

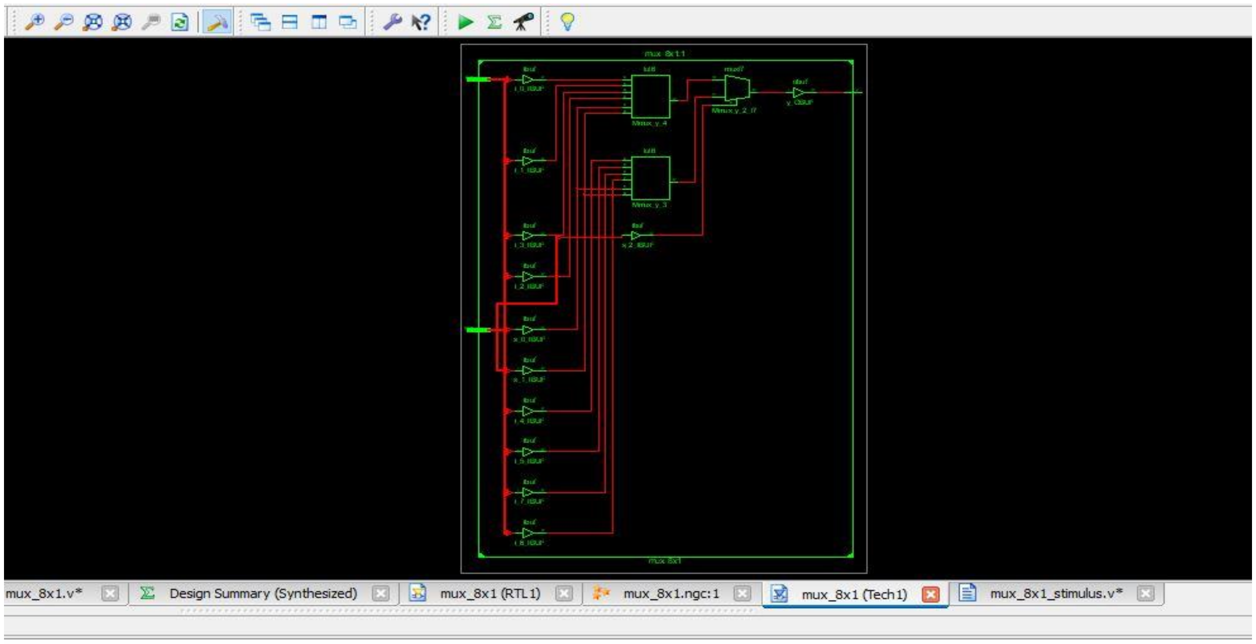
```
module mux_8x1(y,i,s);
output y;
input [7:0]i;
input [2:0]s;
reg y;
// Behavioural
always@(i or s)
// Using case statement
case(s)
3'b000:y = i[0];
3'b001:y = i[1];
3'b010:y = i[2];
3'b011:y = i[3];
3'b100:y = i[4];
3'b101:y = i[5];
3'b110:y = i[6];
3'b111:y = i[7];
endcase
endmodule
```

Test Bench

```
module mux_8x1_stimulus;
        // Inputs
        reg [7:0] i;
        reg [2:0] s;
        // Outputs
        wire y;
        // Instantiate the Unit Under Test (UUT)
        mux_8x1 uut (
                .y(y),
                .i(i),
                .s(s));
        initial
        begin
                // Initialize Inputs
                //$monitor($time,"y=%b,i=%b,s=%b",y,i,s);
                i = 0;
                s = 0;
                end
        always # 10 i=i+1;
        always # 10 s=s+1;
endmodule
```

RTL





Example # 4 {2_to_4} Decoder (Behavioral modelling)

Code

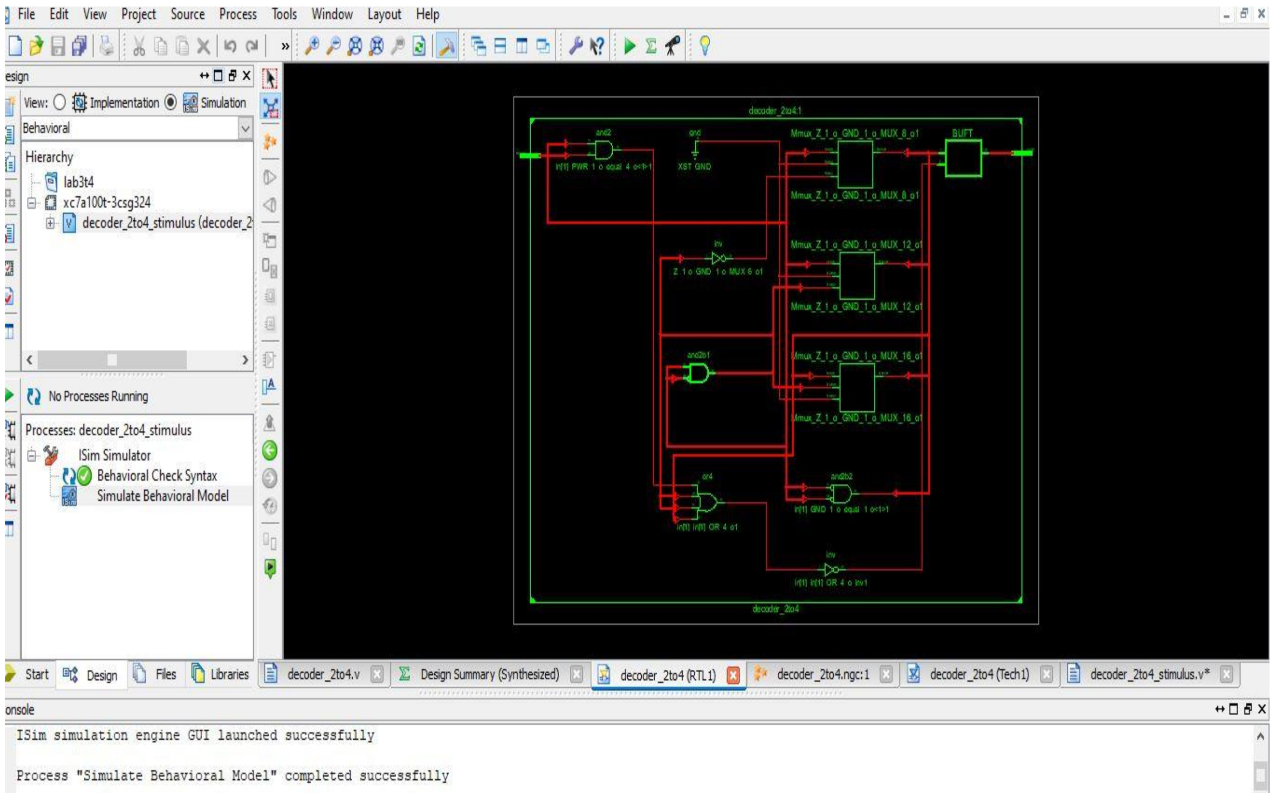
```
module decoder_2to4(in,y);
input [1:0]in;
output reg[3:0]y;
always@(in)
begin
        if(in==2'b00)
  y = 4'b0001;
        else if(in==2'b01)
  y = 4'b0010;
        else if(in==2'b10)
  y = 4'b0100;
        else if(in==2'b11)
  y = 4'b1000;
        else
  y = 4'bzzzz;
        end
```

```
endmodule
```

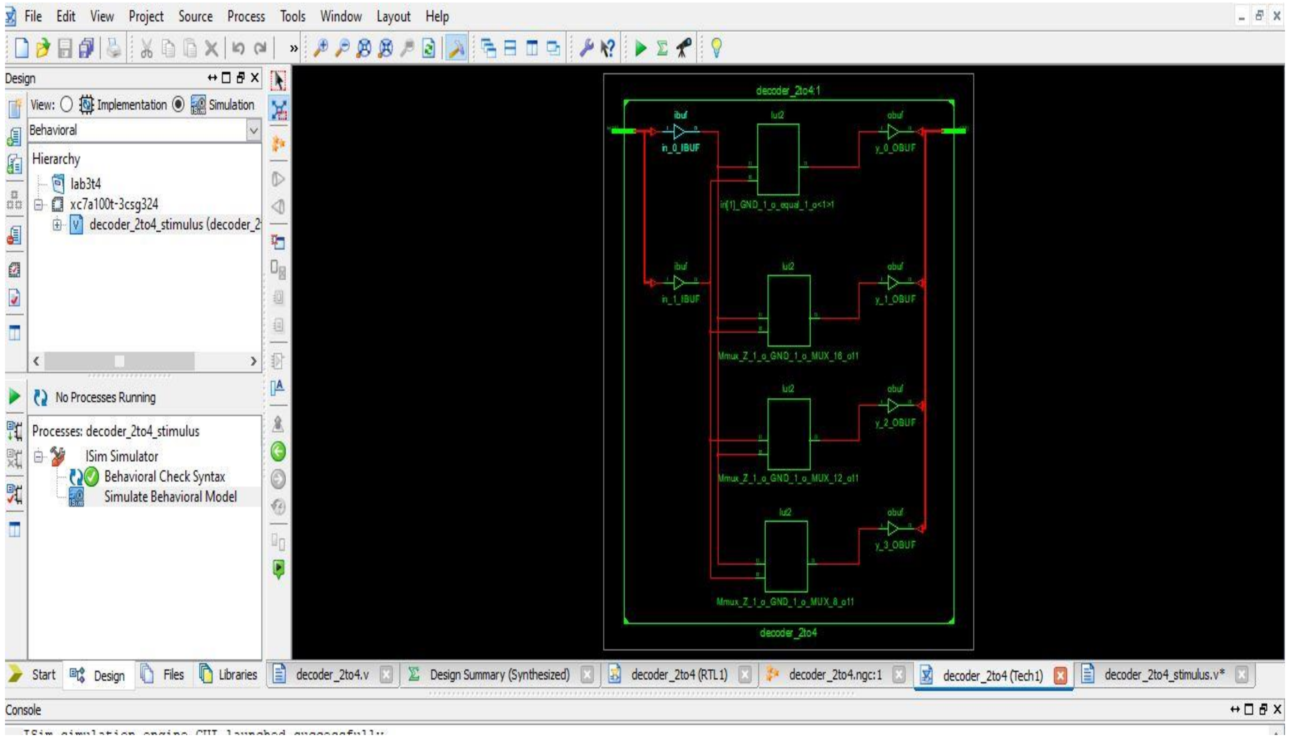
Test Bench

```
module decoder_2to4_stimulus;

        // Inputs
        reg [1:0] in;

        // Outputs
        wire [3:0] y;

        // Instantiate the Unit Under Test (UUT)
        decoder_2to4 uut (
                .in(in),
                .y(y)
        );

        initial
        begin
        //$monitor($time , "in=%b,y=%b",in,y);
                // Initialize Inputs
      in = 2'b00;
#4    in = 2'b01;
#4    in = 2'b10;
#4    in = 2'b11;
#2    $finish;
end
endmodule
```

RTL

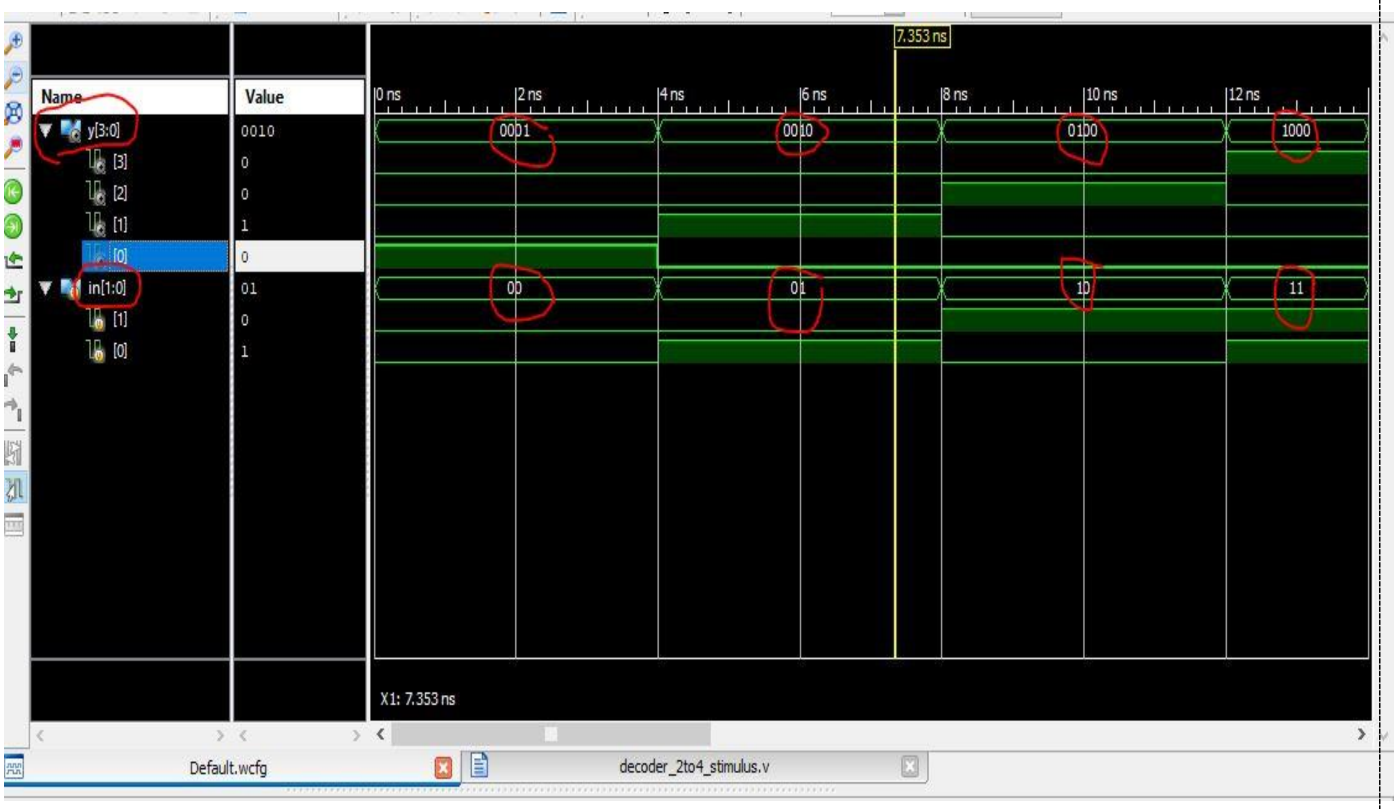


```
ISim simulation engine GUI launched successfully

Process "Simulate Behavioral Model" completed successfully
```



```
ISim simulation engine GUI launched successfully
```

Waveform


7.353 ns
Name
Value
y[3:0]
[3]
[2]
[1]
[0]
in[1:0]
[1]
[0]
0010
0
0
1
0
01
0
1
0 ns
2 ns
4 ns
6 ns
8 ns
10 ns
12 ns
0001
0010
0100
1000
00
01
10
11
X1: 7.353 ns
Default.wcfg
decoder_2to4_stimulus.v

ALU

Code

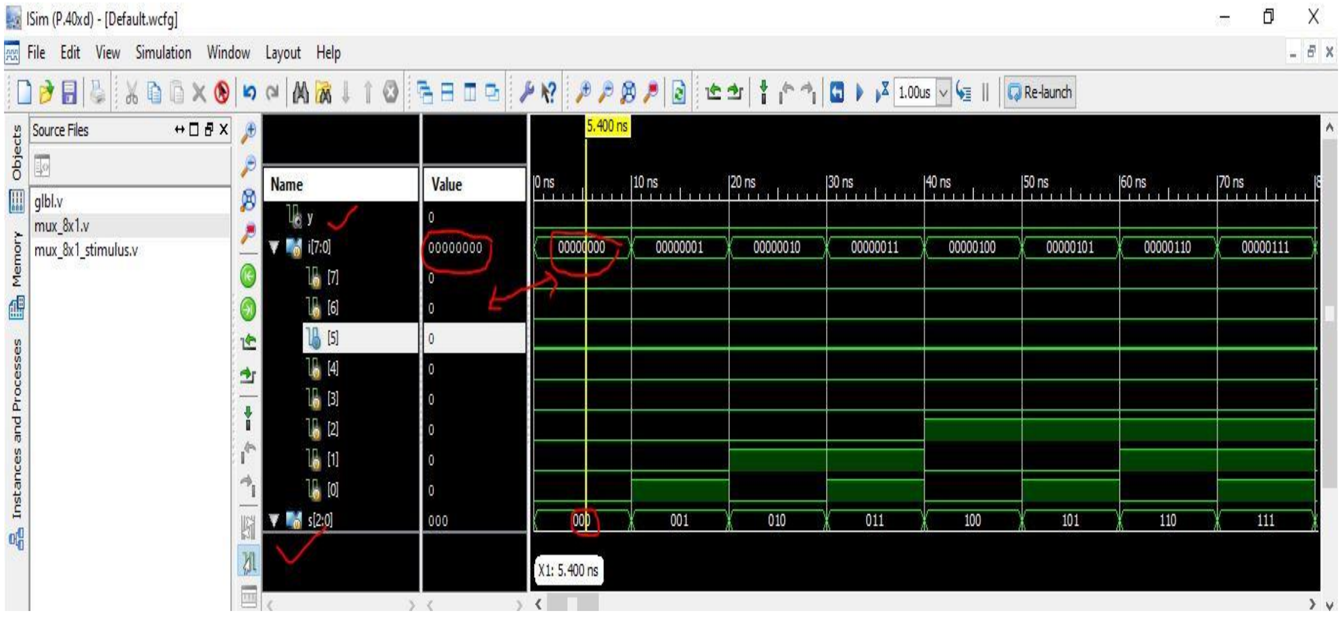
```
module alu(c,a,b,sel);
input [8:0]a,b;
input [3:0]sel;
output [8:0]c;
reg [8:0]c;

always @(sel,a,b)
begin
case(sel)
4'b0000: c = a+b;
4'b0001: c = a-b;
4'b0010: c = a*b;
4'b0011: c = a&&b;
4'b0100: c = a||b;
4'b0101: c = a&b;
4'b0100: c = a|b;
4'b0111: c = a<<1;
4'b1000: c = a>>b;
4'b1001: c = ~a;
4'b1010: c = !a;
4'b1011: c = a+1;
4'b1100: c = a-1;
4'b1101: c = b+1;
4'b1110: c = b-1;
4'b1111: c = b;
endcase
end
endmodule
```

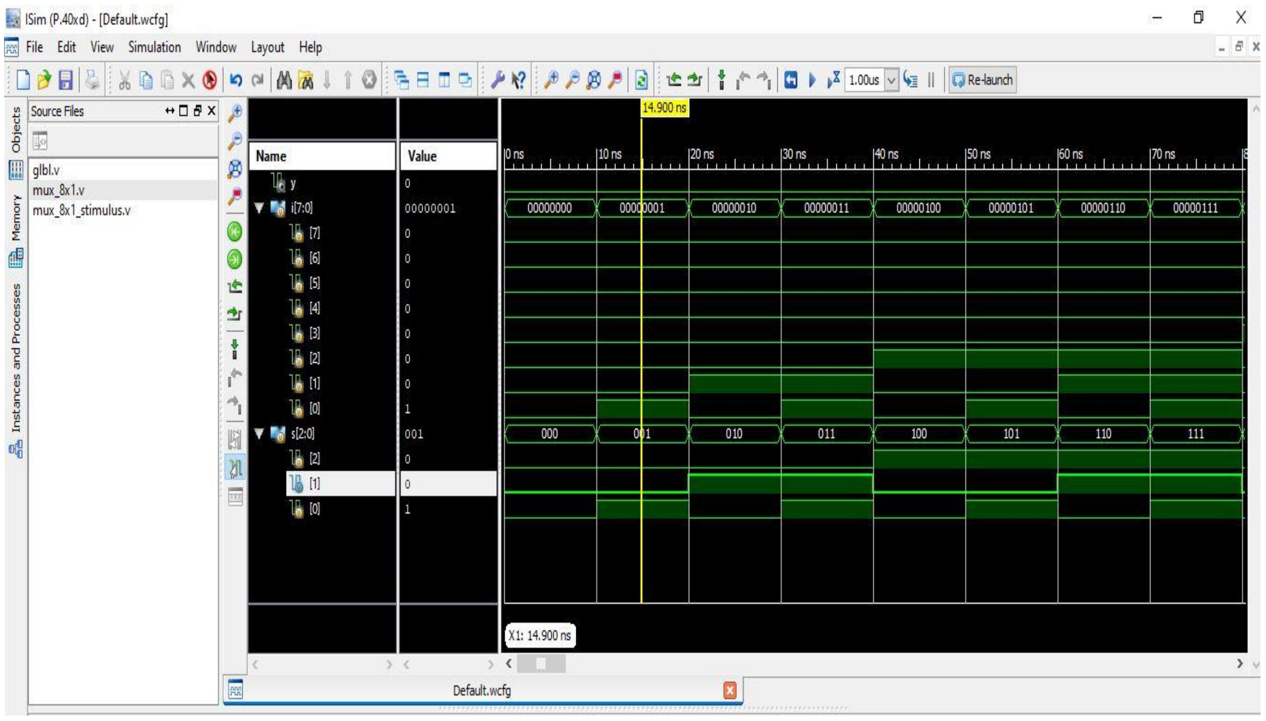
Test Bench

```
module alu_stimulus;
        reg [8:0] a;
        reg [8:0] b;
        reg [3:0] sel;
        // Outputs
        wire [7:0] c;
        // Instantiate the Unit Under Test (UUT)
        alu uut (
                .c(c),
                .a(a),
                .b(b),
                .sel(sel));
        initial
        begin
                // Initialize Inputs
      a = 1;b = 1; sel = 4'b0000;
#1    a = 1;b = 0; sel = 4'b0001;
#1    a = 1;b = 1; sel = 4'b0010;
#1    a = 1;b = 1; sel = 4'b0011;
#1    a = 0;b = 1; sel = 4'b0100;
#1    a = 1;b = 0; sel = 4'b0101;
#1    a = 1;b = 1; sel = 4'b0110;
#1    a = 1;b = 1; sel = 4'b0111;
#1    a = 1;b = 1; sel = 4'b1000;
#1    a = 0;b = 1; sel = 4'b1001;
#1    a = 1;b = 0; sel = 4'b1010;
#1    a = 1;b = 1; sel = 4'b1011;
#1    a = 1;b = 0; sel = 4'b1100;
#1    a = 0;b = 0; sel = 4'b1101;
#1    a = 0;b = 1; sel = 4'b1110;
```

```
#1    a = 0;b = 0; sel = 4'b1111;
#5  $finish;
end
endmodule
```

Waveform





Example ALU

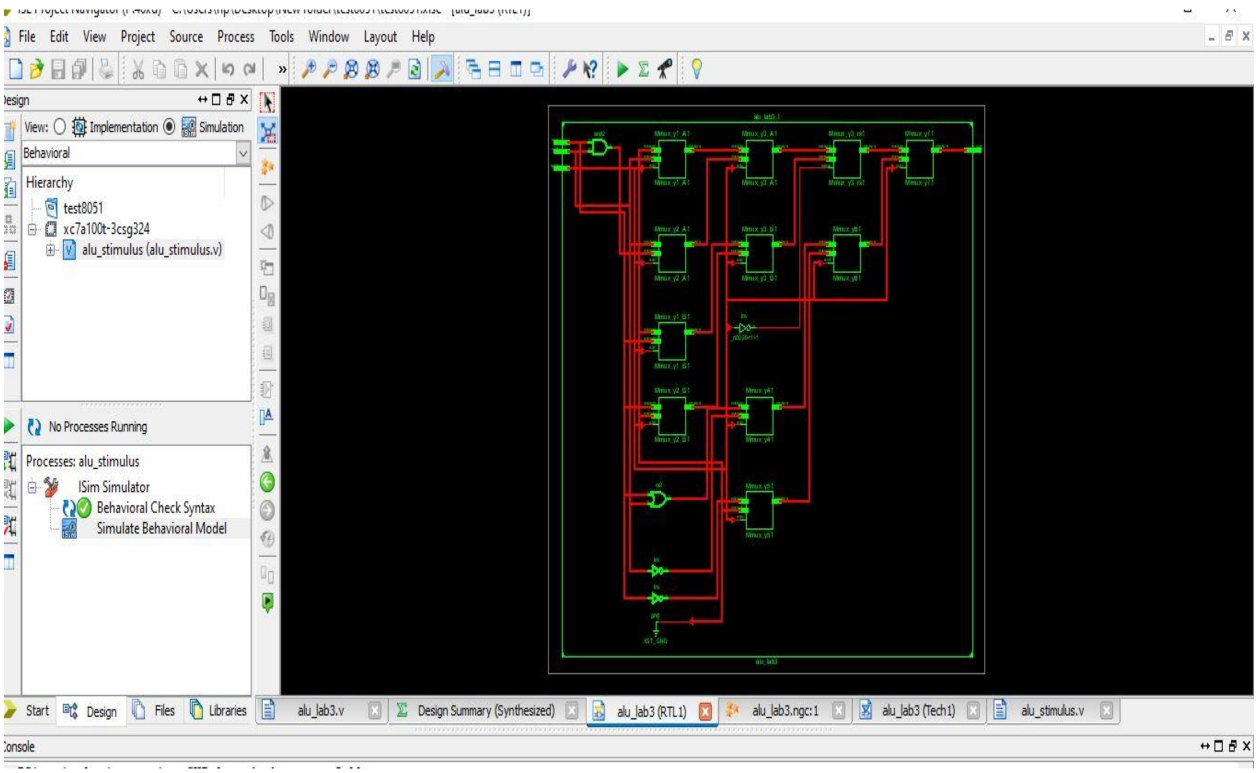
Code

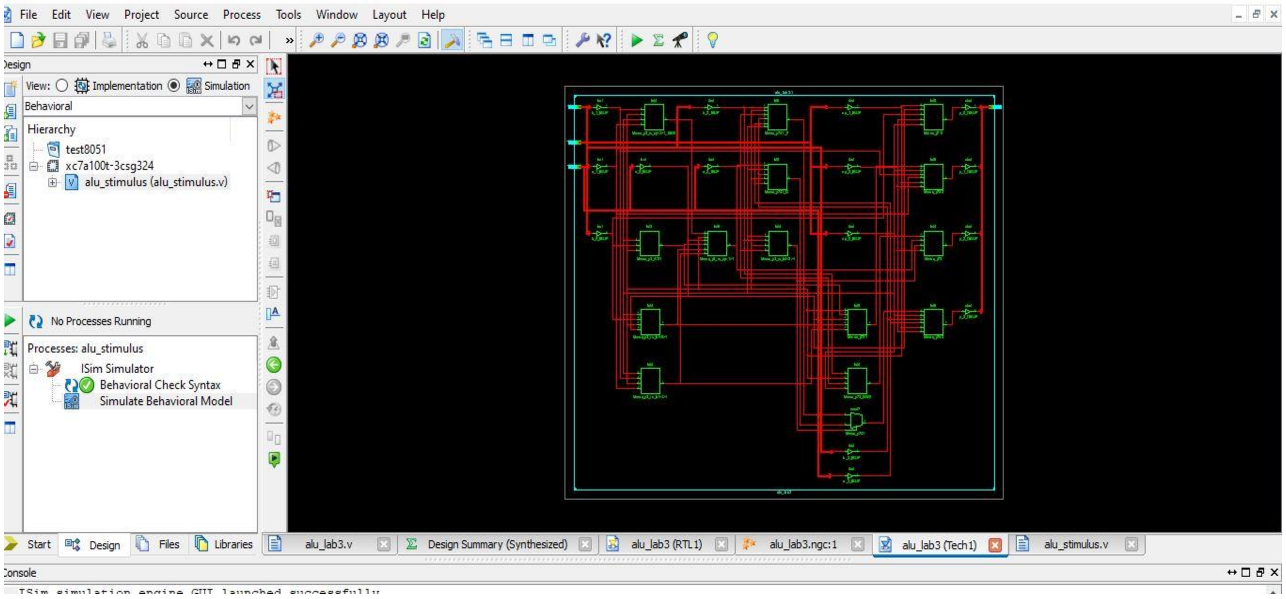
```
module alu_lab3(y,op,a,b);
input[3:0]a,b;
input[2:0]op;
output reg[3:0]y;

always@(*)
begin
case(op)
3'b000:y=a+b;
3'b001:y=a-b;
3'b010:y=a&b;
3'b011:y=a|b;
3'b100:y=~a;
3'b101:y=~(a&b);
3'b110:y=~(a|b);
default:y=0;
endcase
end
endmodule
```

Test Bench

```
module alu_stimulus;
        reg [2:0] op;
        reg [3:0] a;
        reg [3:0] b;
        wire [3:0] y;
        alu_lab3 uut (
                .y(y),
                .op(op),
                .a(a),
                .b(b));
          initial
          begin

        op = 3'b000; a =4'b0010 ; b =4'b0011;
#5      op = 3'b001; a =4'b0011 ; b =4'b0001;
#5      op = 3'b010; a =4'b0111 ; b =4'b0100;
#5      op = 3'b011; a =4'b0111 ; b =4'b0011;
#5      op = 3'b100; a =4'b0111 ; b =4'b0100;
#5      op = 3'b101; a =4'b0110 ; b =4'b0101;
#5      op = 3'b110; a =4'b0100 ; b =4'b0010;
#5      op = 3'b111; a =4'b0101 ; b =4'b0001;
#5      op = 3'b000; a =4'b0100 ; b =4'b0010;
#5      $finish;
end
endmodule
```

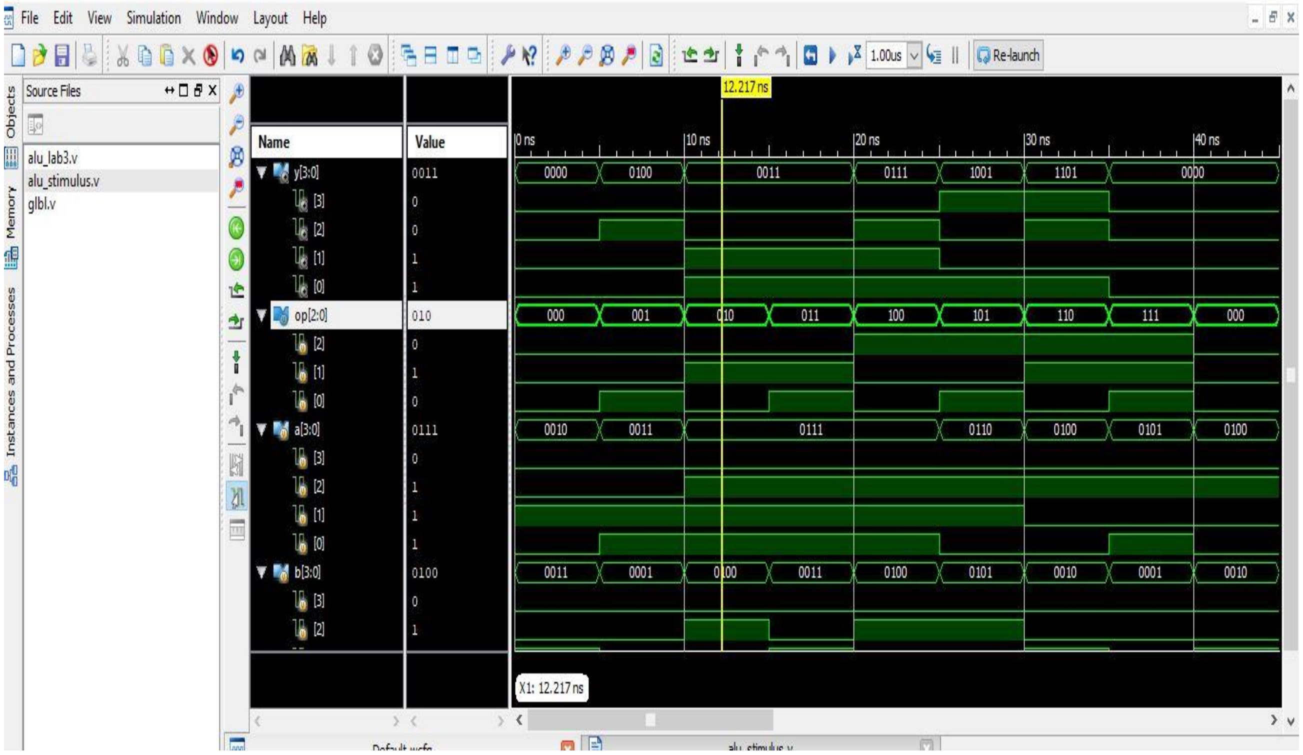
RTL





```
ISim simulation engine GUI launched successfully
```

Waveform



RUBRIC SHEET

Criteria	**Excellent (Full Marks)**	**Good (Partial Marks)**	**Needs Improvement (Low Marks)**	**Max Marks**
Code Writing (Verilog) (**Apply**) (4)	Writes fully optimized and error-free Verilog code for combinational and sequential circuits independently.	Writes mostly correct code with minor errors, requiring limited guidance.	The code contains major errors or is incomplete, requiring significant help to correct.	**/4**
Test Bench (**Construct**) (3)	Develops accurate and complete test benches to simulate and verify circuit functionality without errors.	Creates test benches with minor issues that require small corrections.	Fails to create a proper test bench or is unable to verify functionality.	**/3**
FPGA Implementation (**Construct & Apply**) (3)	Successfully implements the design on the FPGA trainer board, demonstrating correct and reliable hardware performance.	Implement the design with minor issues that require troubleshooting.	Unable to correctly implement the design on the FPGA, or the hardware fails.	**/3**
Design Objectives (**Apply & Construct**) (3)	Clearly states and applies theoretical concepts to fully achieve all given design objectives.	States and applies concepts partially, achieving some design objectives.	Fails to state or apply theoretical concepts, achieving none or minimal objectives.	**/3**
Viva (**Collaborate**) (2)	Communicates ideas clearly and confidently, demonstrating full understanding of concepts and the design process.	Communicates with some hesitation or minor conceptual gaps.	Struggles to explain concepts or answer questions correctly.	**/2**
Total				**/15**

Name: ________________ Instructor Signature: ___________________

Reg #: ___________________ Date: _________________