Nick Ji: A18060321
Josue Galindo: A18097946
ECE 25
Section: A53
Wednesday, 3p – 6p, WLH 2211

Lab 5: A calculator in action

Introduction:

The purpose of this lab experiment is to build a calculator that inputs numbers from the Sony TV Remote on the 7-segment display which adds another number when inputted again, using the results from the previous labs.

Procedure:

1. Open the given file lab5top.v.
2. Cascade the full adder from lab 2 to make a 4-bit adder.
3. To do so, consecutively connect the carry output from each FA to the next one.
4. The result bits from each FA will be a bit for the 4-bit number we need.
5. Write the simulation code for 4 FAs together forming a 4-bit adder.
6. Assign 4 bit numbers directly to the inputs and observe the results of simulation.
7. Edit the given file lab5top.v to match the modules from the previous labs.
8. Verify the correctness of the edited lab5top.v with a TA, simulation is optional.
9. Load the project into a BASYS3 board.
10. On a breadboard, power the Pin 3 of the IR receiver with a 5 V DC source.
11. Connect a 51k Ohm resistor from Pin 1 to the ground.
12. Connect the oscilloscope to the output of the chip, which is Pin 1.
13. Connect a wire from Pin 1 to where the data is mapped on the BASYS3 board.
14. Connect Pin 2 to the ground.
15. Ground the BASYS3 board.
16. Set up the function generator to have 3.9 kHz and 3.3 V peak to peak value.
17. Connect the function generator input to where clk is mapped.
18. Check out with a TA.
19. Briefly describe the circuit and what happens when a button is pressed on the remote and it happens in the circuit.

Circuit (Receiver) :

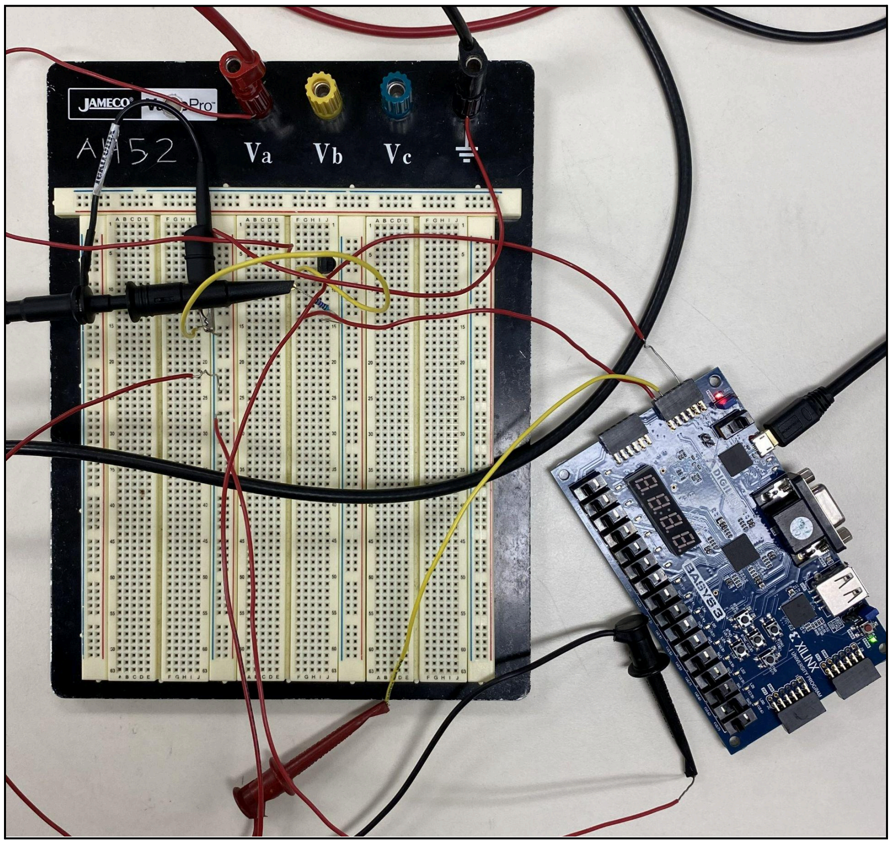


Figure 1. Receiver Circuit (Top View)

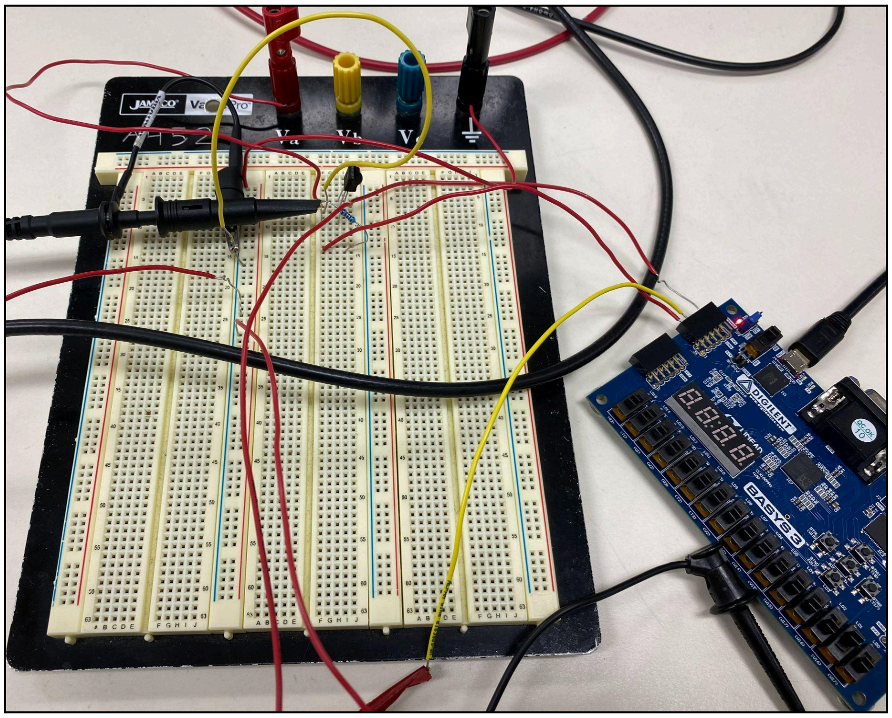
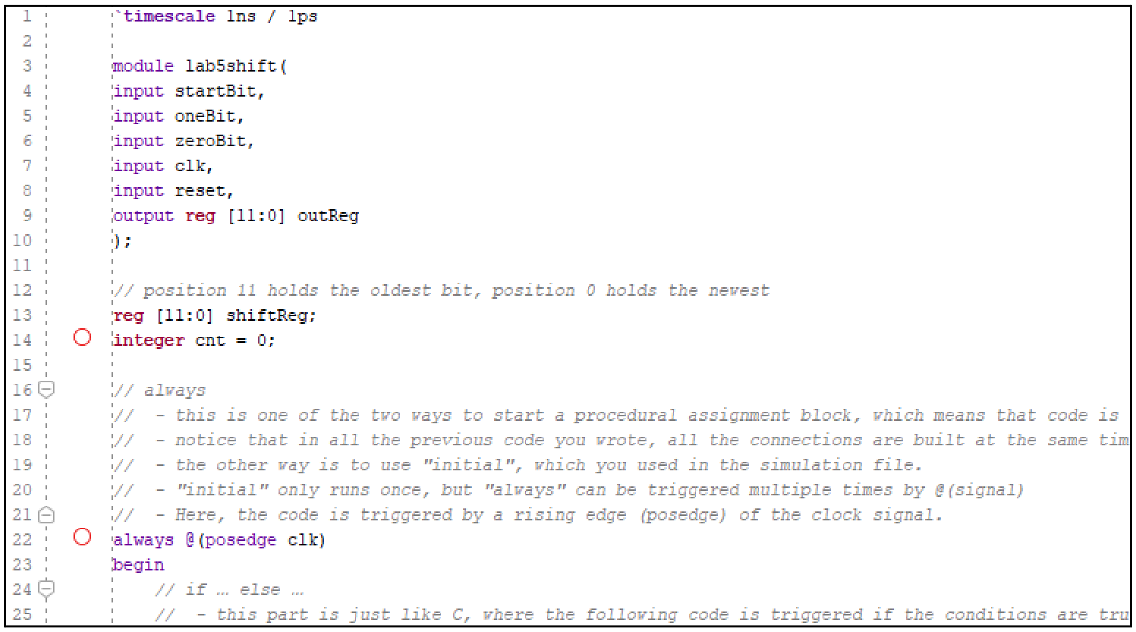
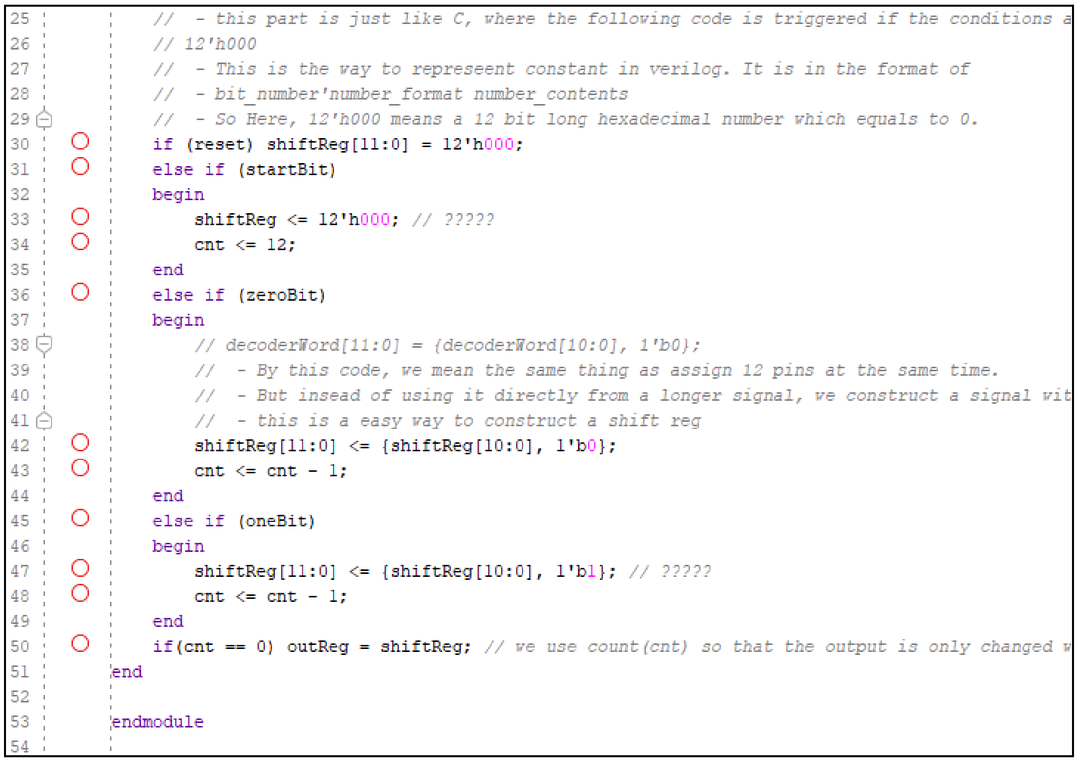


Figure 2. Receiver Circuit (Front View)

Data:

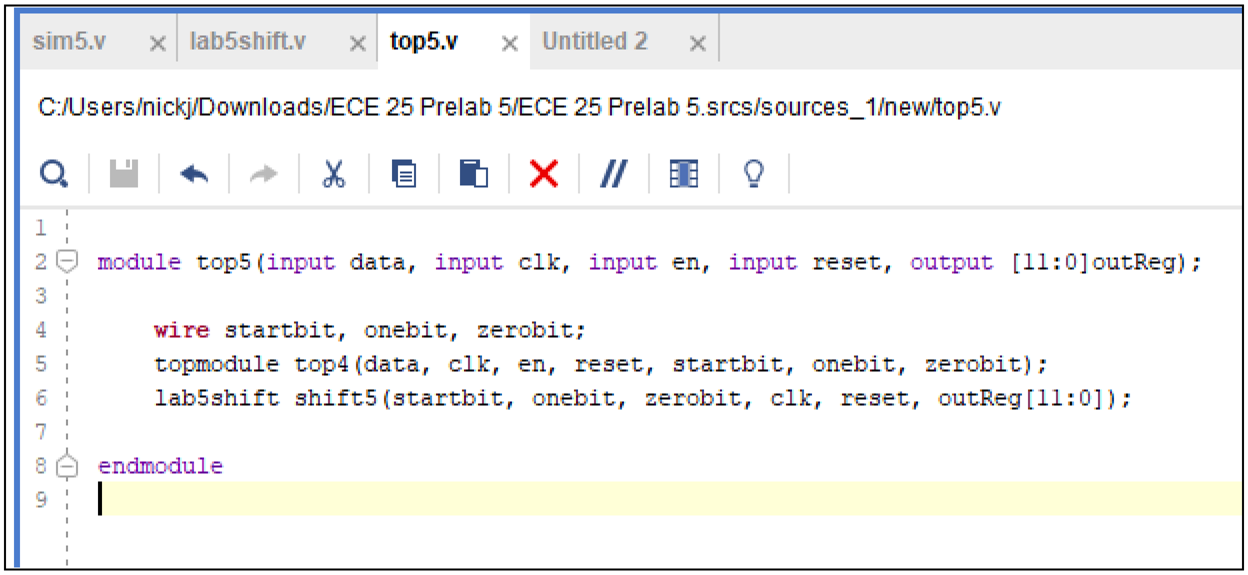


```
`timescale 1ns / 1ps

module lab5shift(
input startBit,
input oneBit,
input zeroBit,
input clk,
input reset,
output reg [11:0] outReg
);

// position 11 holds the oldest bit, position 0 holds the newest
reg [11:0] shiftReg;
integer cnt = 0;

// always
//  - this is one of the two ways to start a procedural assignment block, which means that code is
//  - notice that in all the previous code you wrote, all the connections are built at the same tim
//  - the other way is to use "initial", which you used in the simulation file.
//  - "initial" only runs once, but "always" can be triggered multiple times by @(signal)
//  - Here, the code is triggered by a rising edge (posedge) of the clock signal.
always @(posedge clk)
begin
    // if … else …
    //  - this part is just like C, where the following code is triggered if the conditions are tru
```



```
    //  - this part is just like C, where the following code is triggered if the conditions a
    // 12'h000
    //  - This is the way to represeent constant in verilog. It is in the format of
    //  - bit_number'number_format number_contents
    //  - So Here, 12'h000 means a 12 bit long hexadecimal number which equals to 0.
    if (reset) shiftReg[11:0] = 12'h000;
    else if (startBit)
    begin
        shiftReg <= 12'h000; // ?????
        cnt <= 12;
    end
    else if (zeroBit)
    begin
        // decoderWord[11:0] = {decoderWord[10:0], 1'b0};
        //  - By this code, we mean the same thing as assign 12 pins at the same time.
        //  - But insead of using it directly from a longer signal, we construct a signal wit
        //  - this is a easy way to construct a shift reg
        shiftReg[11:0] <= {shiftReg[10:0], 1'b0};
        cnt <= cnt - 1;
    end
    else if (oneBit)
    begin
        shiftReg[11:0] <= {shiftReg[10:0], 1'b1}; // ?????
        cnt <= cnt - 1;
    end
    if(cnt == 0) outReg = shiftReg; // we use count(cnt) so that the output is only changed w
end

endmodule
```

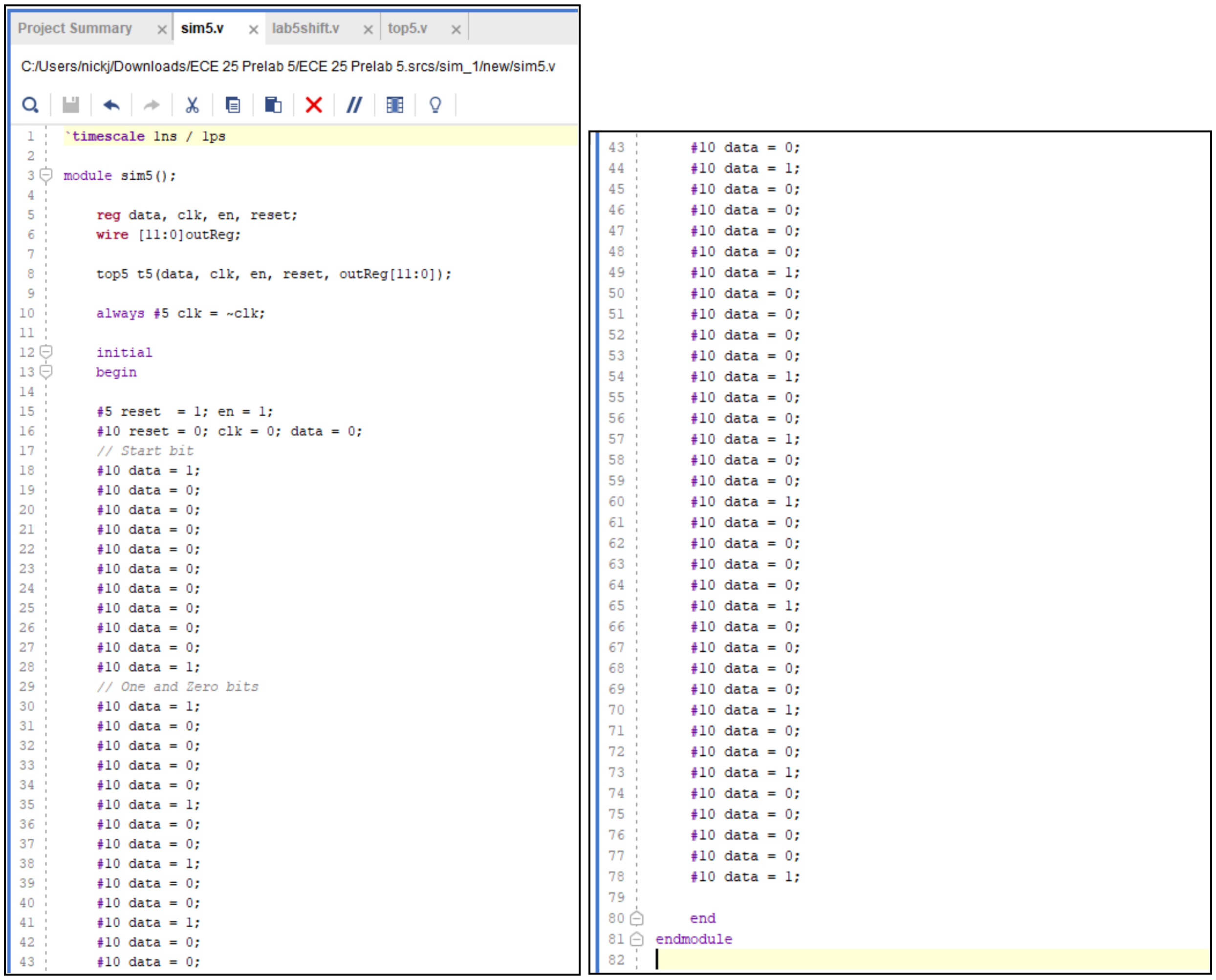
Figure 3. lab5shift.v


sim5.v lab5shift.v top5.v Untitled 2

C:/Users/nickj/Downloads/ECE 25 Prelab 5/ECE 25 Prelab 5.srcs/sources_1/new/top5.v

```

module top5(input data, input clk, input en, input reset, output [11:0]outReg);

    wire startbit, onebit, zerobit;
    topmodule top4(data, clk, en, reset, startbit, onebit, zerobit);
    lab5shift shift5(startbit, onebit, zerobit, clk, reset, outReg[11:0]);

endmodule

```

Figure 4. Lab 5 Top Module


Project Summary sim5.v lab5shift.v top5.v

C:/Users/nickj/Downloads/ECE 25 Prelab 5/ECE 25 Prelab 5.srcs/sim_1/new/sim5.v

```
`timescale 1ns / 1ps

module sim5();

    reg data, clk, en, reset;
    wire [11:0]outReg;

    top5 t5(data, clk, en, reset, outReg[11:0]);

    always #5 clk = ~clk;

    initial
    begin

    #5 reset  = 1; en = 1;
    #10 reset = 0; clk = 0; data = 0;
    // Start bit
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    // One and Zero bits
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 0;
    #10 data = 1;

    end
endmodule

```

Figure 5. Lab 5 Top Module Simulation Code

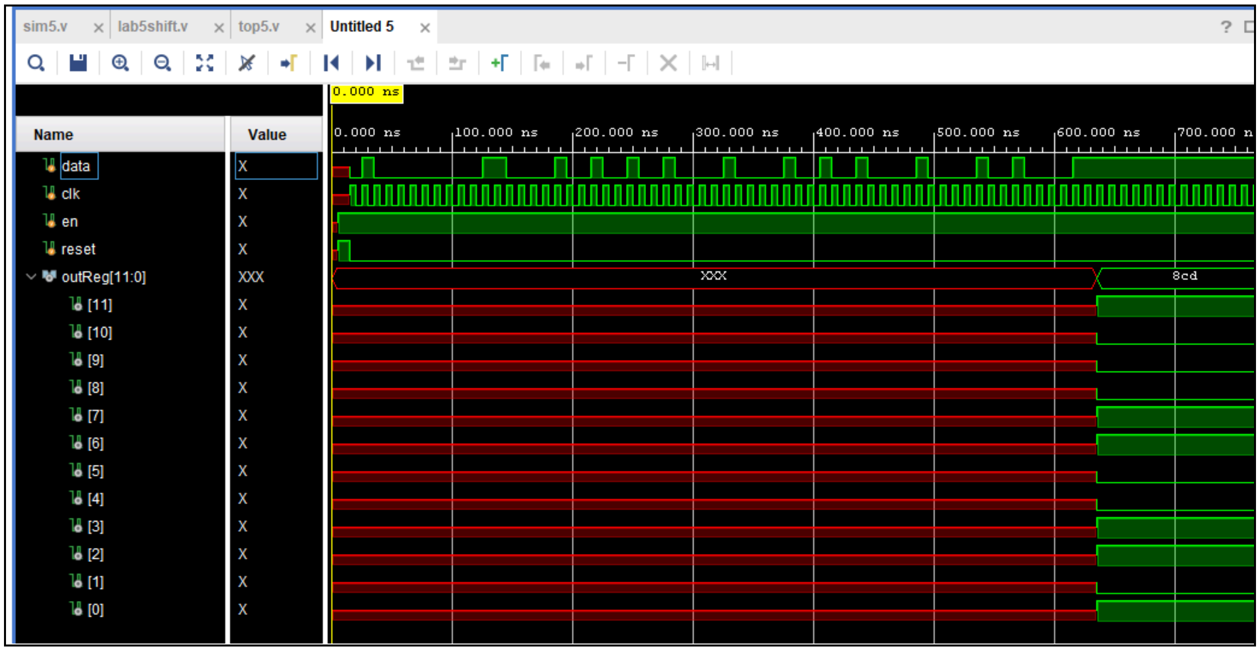
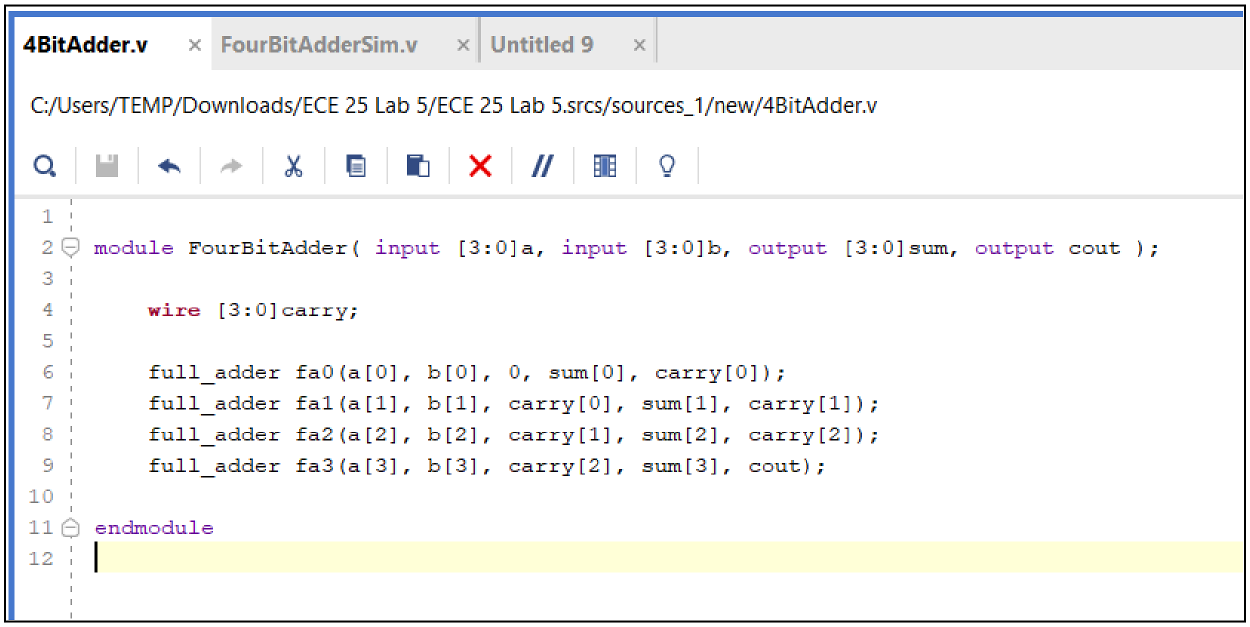


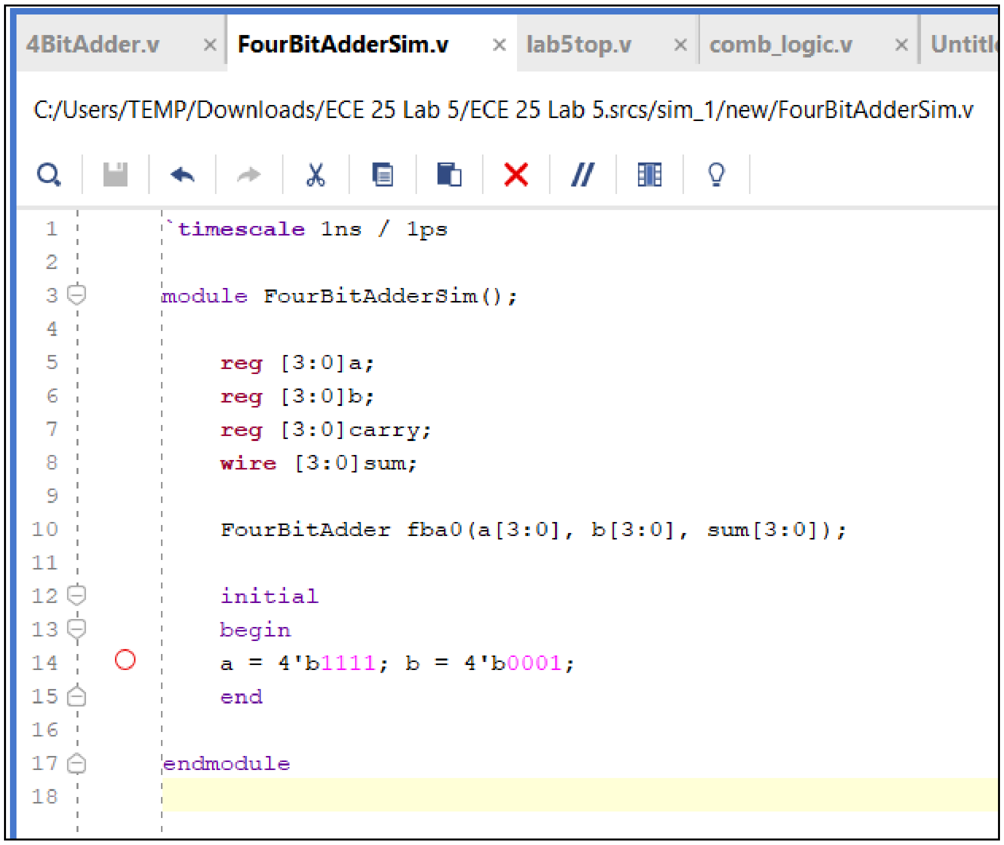
Figure 6. Lab 5 Top Module Simulation



C:/Users/TEMP/Downloads/ECE 25 Lab 5/ECE 25 Lab 5.srcs/sources_1/new/4BitAdder.v

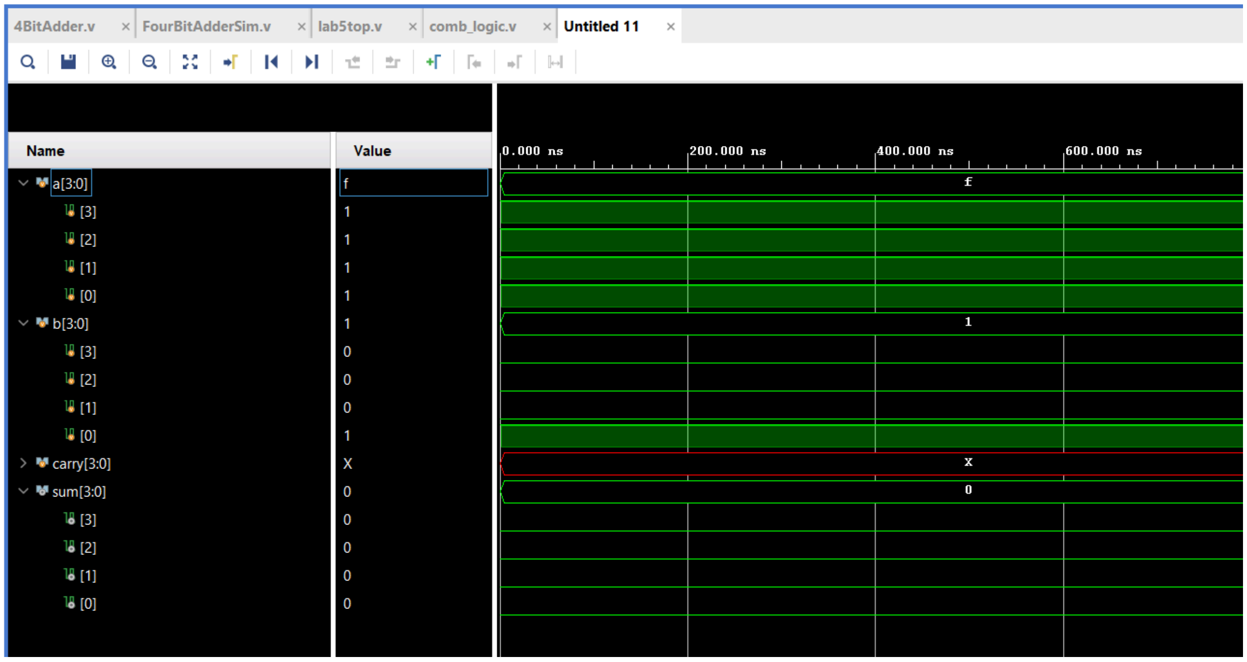
```

module FourBitAdder( input [3:0]a, input [3:0]b, output [3:0]sum, output cout );

    wire [3:0]carry;

    full_adder fa0(a[0], b[0], 0, sum[0], carry[0]);
    full_adder fa1(a[1], b[1], carry[0], sum[1], carry[1]);
    full_adder fa2(a[2], b[2], carry[1], sum[2], carry[2]);
    full_adder fa3(a[3], b[3], carry[2], sum[3], cout);

endmodule

```

Figure 7. 4-Bit Adder Module


4BitAdder.v
FourBitAdderSim.v
lab5top.v
comb_logic.v
Untitl
C:/Users/TEMP/Downloads/ECE 25 Lab 5/ECE 25 Lab 5.srcs/sim_1/new/FourBitAdderSim.v

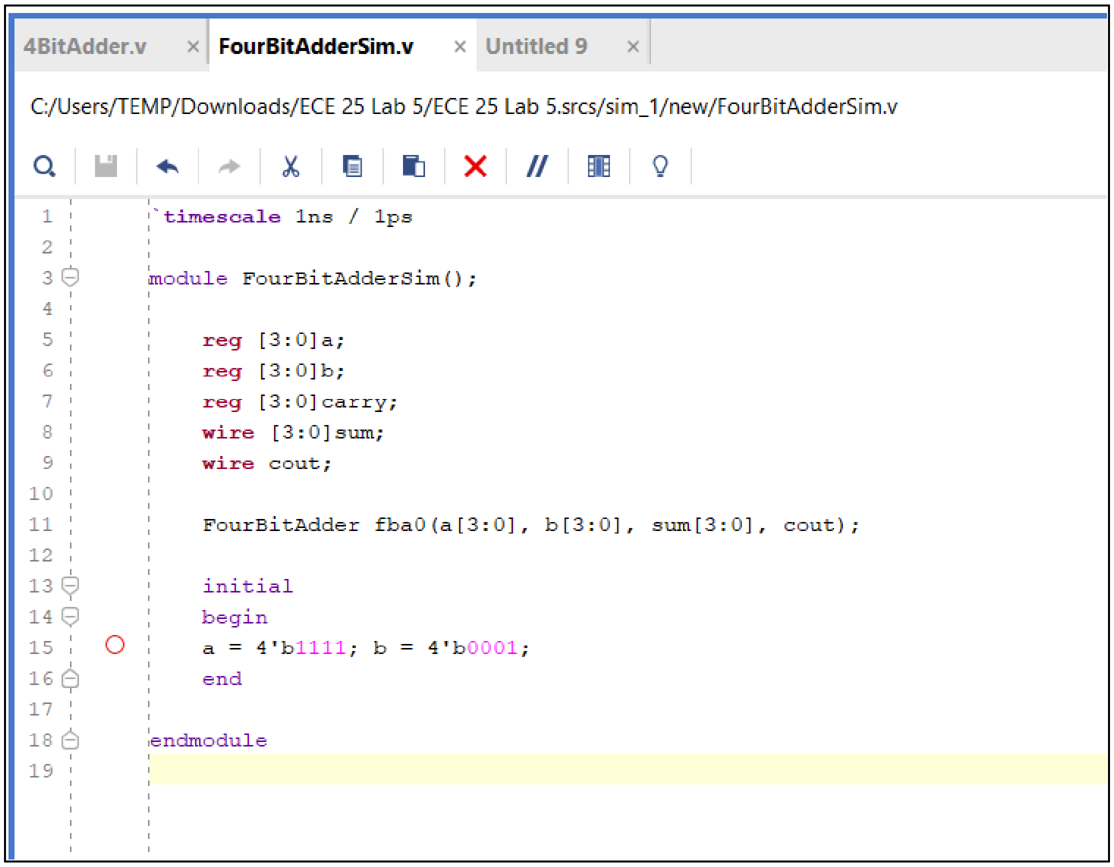
```
`timescale 1ns / 1ps

module FourBitAdderSim();

    reg [3:0]a;
    reg [3:0]b;
    reg [3:0]carry;
    wire [3:0]sum;

    FourBitAdder fba0(a[3:0], b[3:0], sum[3:0]);

    initial
    begin
    a = 4'b1111; b = 4'b0001;
    end

endmodule
```

Figure 8. 4-Bit Adder Sim Code



Name	Value
a[3:0]	f
[3]	1
[2]	1
[1]	1
[0]	1
b[3:0]	1
[3]	0
[2]	0
[1]	0
[0]	1
carry[3:0]	X
sum[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0

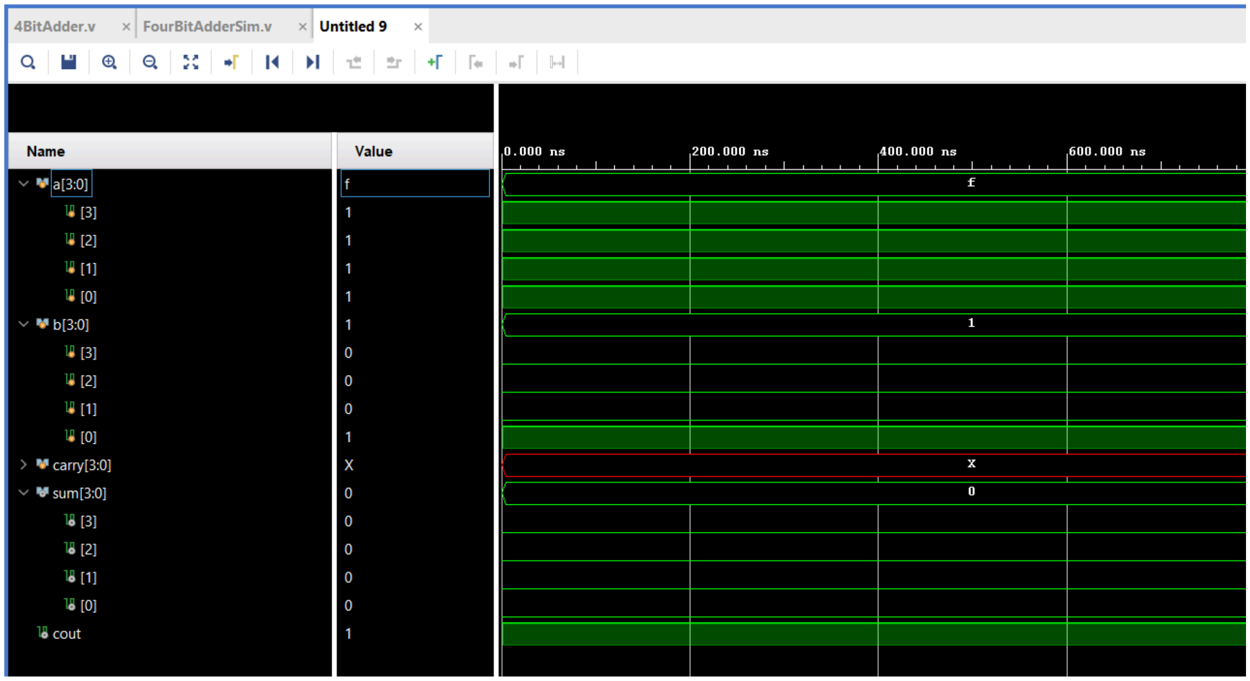
Figure 9. 4-Bit Adder Simulation



C:/Users/TEMP/Downloads/ECE 25 Lab 5/ECE 25 Lab 5.srcs/sim_1/new/FourBitAdderSim.v

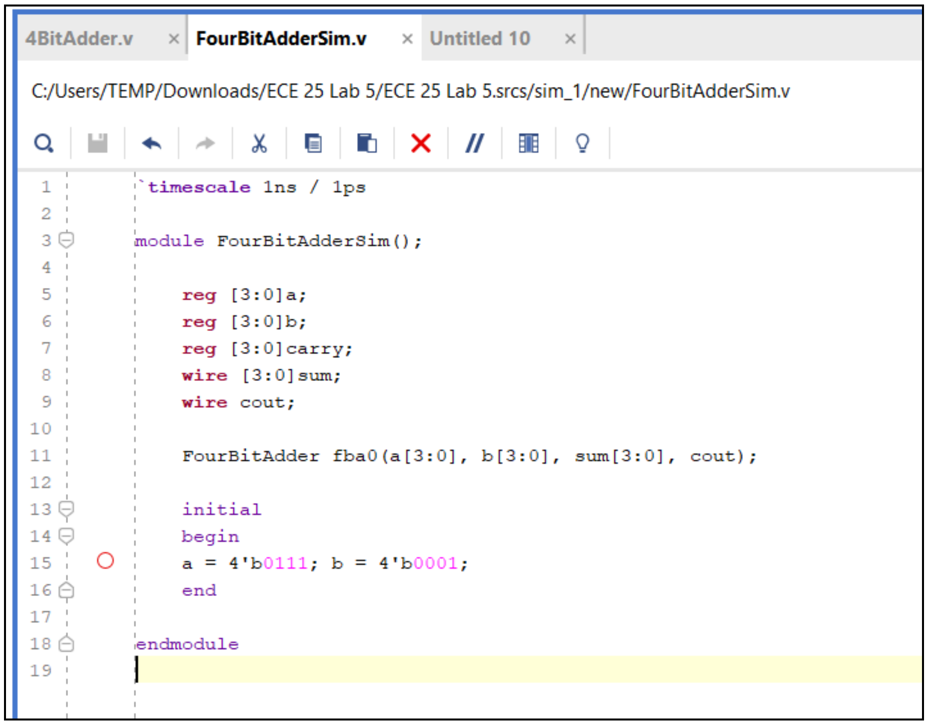
```
`timescale 1ns / 1ps

module FourBitAdderSim();

    reg [3:0]a;
    reg [3:0]b;
    reg [3:0]carry;
    wire [3:0]sum;
    wire cout;

    FourBitAdder fba0(a[3:0], b[3:0], sum[3:0], cout);

    initial
    begin
    a = 4'b1111; b = 4'b0001;
    end

endmodule
```

Figure 10. 4-Bit Adder Sim Code W/ Carryout Case 1



Name	Value
a[3:0]	f
[3]	1
[2]	1
[1]	1
[0]	1
b[3:0]	1
[3]	0
[2]	0
[1]	0
[0]	1
carry[3:0]	X
sum[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0
cout	1

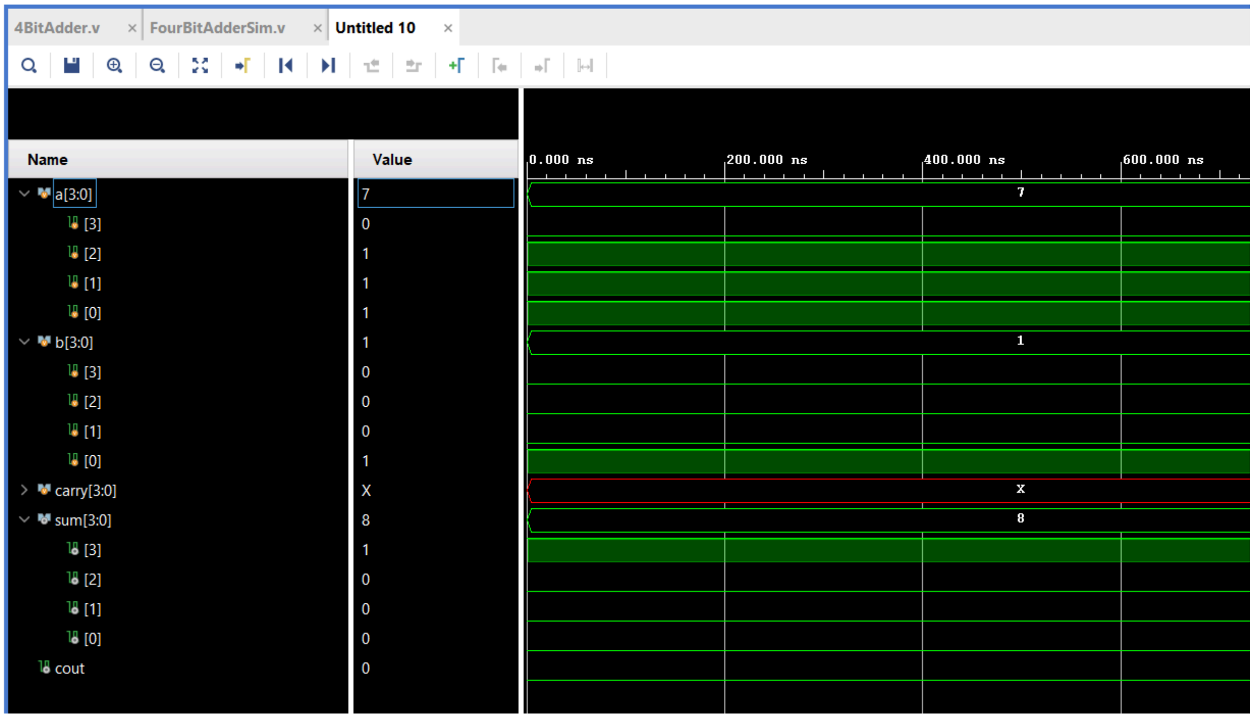
Figure 11. 4-Bit Adder Sim W/ Carryout Case 1



C:/Users/TEMP/Downloads/ECE 25 Lab 5/ECE 25 Lab 5.srcs/sim_1/new/FourBitAdderSim.v

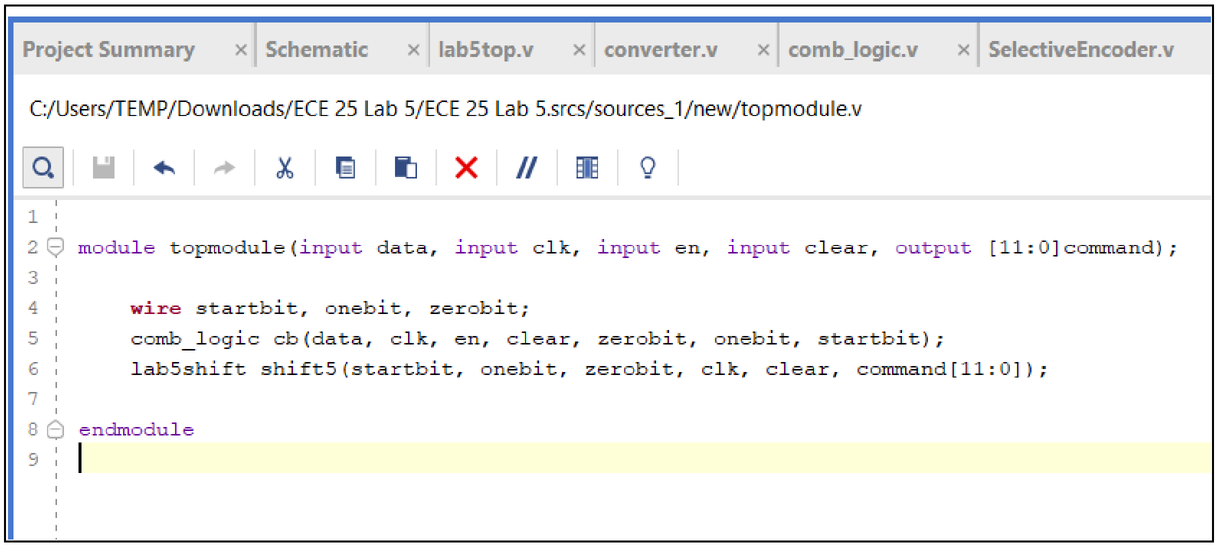
```
`timescale 1ns / 1ps

module FourBitAdderSim();

    reg [3:0]a;
    reg [3:0]b;
    reg [3:0]carry;
    wire [3:0]sum;
    wire cout;

    FourBitAdder fba0(a[3:0], b[3:0], sum[3:0], cout);

    initial
    begin
    a = 4'b0111; b = 4'b0001;
    end

endmodule
```

Figure 12. 4-Bit Adder Sim Code W/ Carryout Case 2



Name	Value
a[3:0]	7
[3]	0
[2]	1
[1]	1
[0]	1
b[3:0]	1
[3]	0
[2]	0
[1]	0
[0]	1
carry[3:0]	X
sum[3:0]	8
[3]	1
[2]	0
[1]	0
[0]	0
cout	0

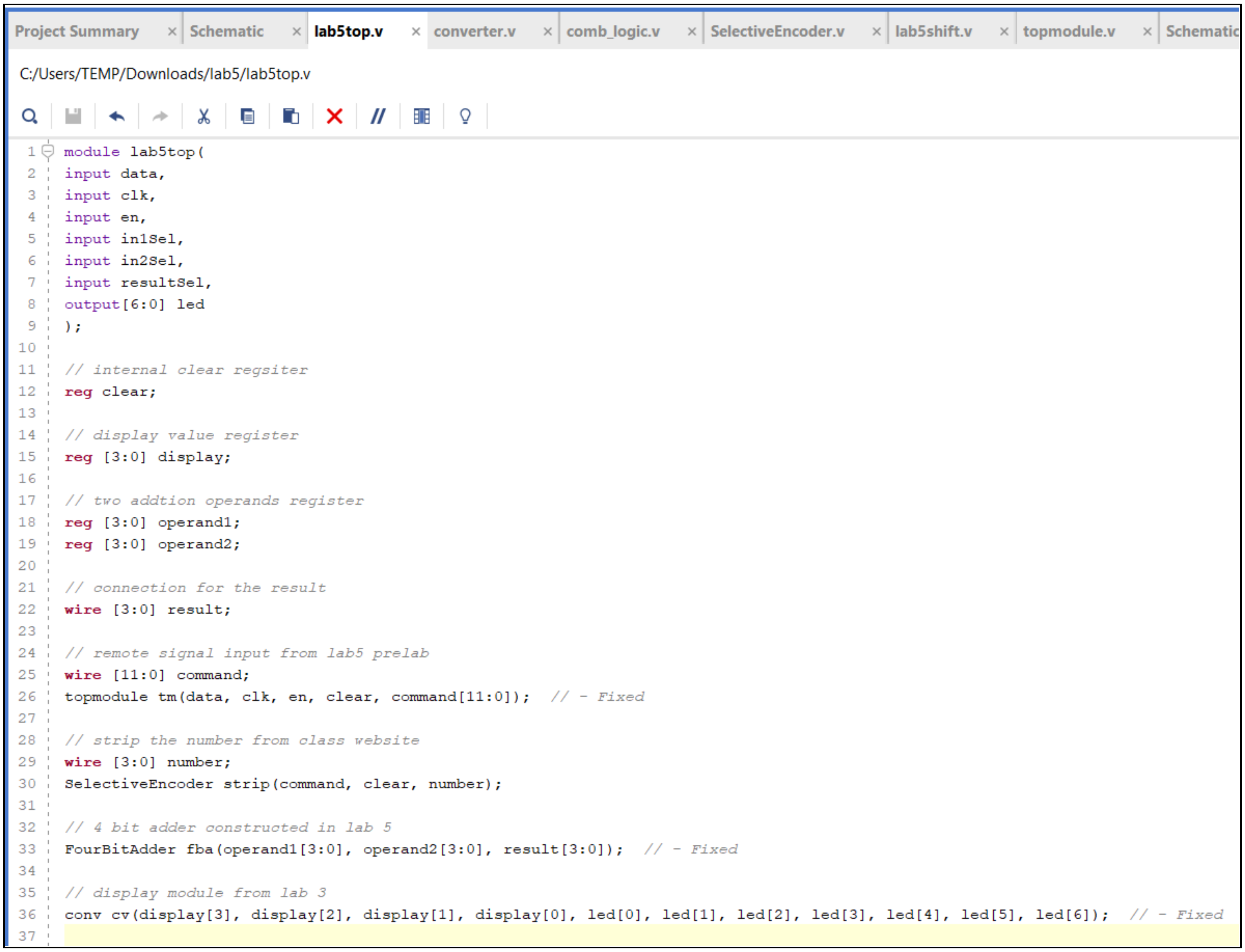
Figure 13. 4-Bit Adder Sim W/ Carryout Case 2



C:/Users/TEMP/Downloads/ECE 25 Lab 5/ECE 25 Lab 5.srcs/sources_1/new/topmodule.v

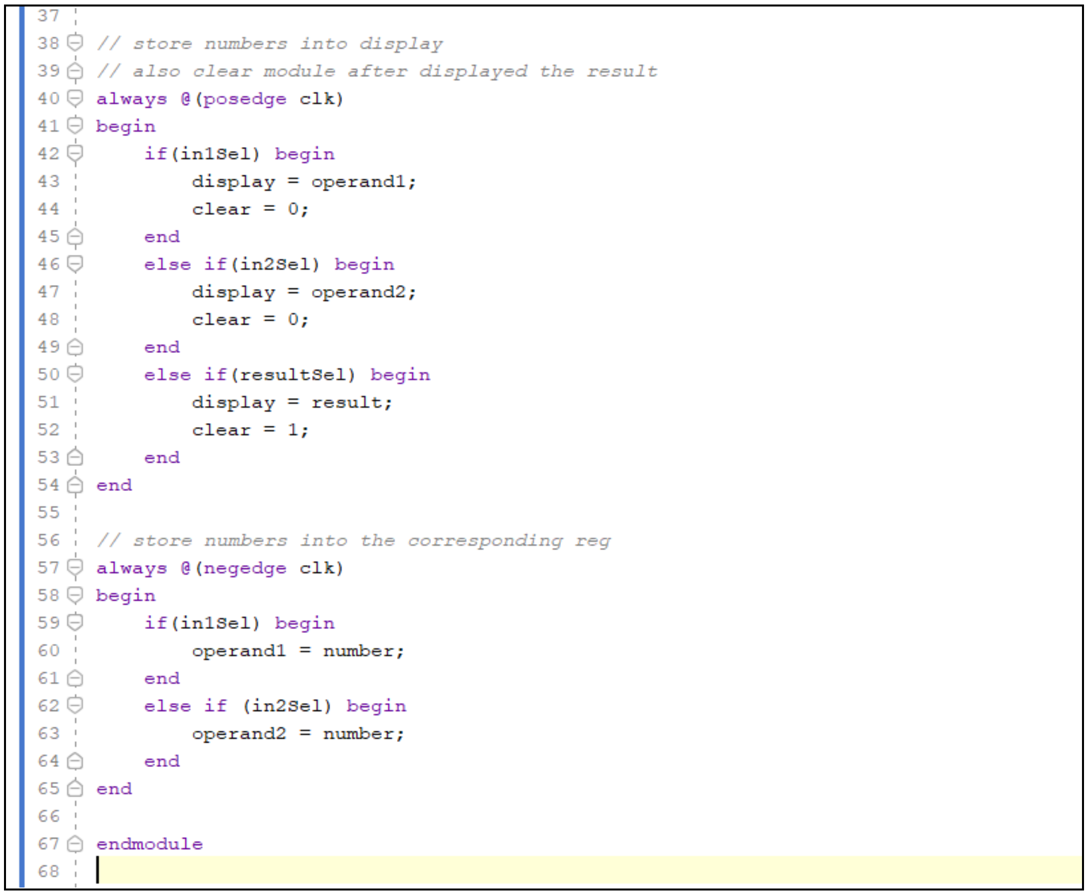
```verilog

module topmodule(input data, input clk, input en, input clear, output [11:0]command);

    wire startbit, onebit, zerobit;
    comb_logic cb(data, clk, en, clear, zerobit, onebit, startbit);
    lab5shift shift5(startbit, onebit, zerobit, clk, clear, command[11:0]);

endmodule

```

Figure 14. Top Module for Shift Registers



C:/Users/TEMP/Downloads/lab5/lab5top.v

```
module lab5top(
input data,
input clk,
input en,
input in1Sel,
input in2Sel,
input resultSel,
output[6:0] led
);

// internal clear regsiter
reg clear;

// display value register
reg [3:0] display;

// two addtion operands register
reg [3:0] operand1;
reg [3:0] operand2;

// connection for the result
wire [3:0] result;

// remote signal input from lab5 prelab
wire [11:0] command;
topmodule tm(data, clk, en, clear, command[11:0]);  // - Fixed

// strip the number from class website
wire [3:0] number;
SelectiveEncoder strip(command, clear, number);

// 4 bit adder constructed in lab 5
FourBitAdder fba(operand1[3:0], operand2[3:0], result[3:0]);  // - Fixed

// display module from lab 3
conv cv(display[3], display[2], display[1], display[0], led[0], led[1], led[2], led[3], led[4], led[5], led[6]);  // - Fixed

```



```verilog

// store numbers into display
// also clear module after displayed the result
always @(posedge clk)
begin
    if(in1Sel) begin
        display = operand1;
        clear = 0;
    end
    else if(in2Sel) begin
        display = operand2;
        clear = 0;
    end
    else if(resultSel) begin
        display = result;
        clear = 1;
    end
end

// store numbers into the corresponding reg
always @(negedge clk)
begin
    if(in1Sel) begin
        operand1 = number;
    end
    else if (in2Sel) begin
        operand2 = number;
    end
end

endmodule

```

Figure 15. Edited lab5top.v

Analysis:

By reusing our previous lab's results, we edited the codes and modules to fit into the new top module for this lab. We first understood how the machine will work, which is to receive the data from the TV remote, rearrange the input data with a 12 bit to 4 bit decoder. We then used the 4 bit data to store a number in the register, and displayed it using the 7 segment display on the BASYS3 board. The bits from the input tells the board which segment from a-g will light up. We also built the simple IR Receiver circuit which receives the signal from the TV remote when we press a button. The input signal is then shifted and decoded into 4 bits from 12 bits, and therefore matched with the 7 segments and displayed properly.

At the end, using a 4-bit adder we derived from using half bit adders and full bit adders, we can get the result of the two numbers stored in the register to sum up and display onto the board. To do so, we had to edit the I/O ports to match the input ports of the board to desired ones and also ground the board correctly. We set our switches on the board to function as we needed, for example, enable switch, operand 1, operand 2, and the result switches.

Through the lab, we encountered a few obstacles. We had a problem with identifying the top modules we should use since there are a lot of variables involved in the entire machine. After we tested out and saw that the result didn’t match, we went back and fixed the module to match the right one. We also had problems with displaying the output onto the board. We realized that using an array as output is not acceptable in certain situations. We rewrote our code of the lab5top.v to have the outputs one by one and we were finally able to add the two numbers that were decoded from input bits and display them correctly.

Conclusion:

In conclusion, we successfully built a simple calculator that can receive the input signal from a Sony TV remote, process the signal to store a number and display onto the BASYS3 board’s LEDs, and add another one when we switch the display to result. The entire process is done simultaneously but has a lot of different sections. For example, as the input signal is received, the machine first recognizes the Sony bits. It then decodes what kind of sony bit the signal is, shifts the real bits into a 12-bit shift register. It then constructs the signal that was initially received after all input signals are shifted. After that, using the LEDs, we match the 7 segments to different bits converted from input and display the output onto the board. With the 4-bit adder, we add the two different input signals and display onto the board when it’s switched to. Overall, the experiment is completed as expected.