Class:	CPE300L		Semester:	Fall 2021
Points		Document author:	Jerrod Batu	
		Author's email:	batuj1@unlv.nevada.edu	
		Document topic:	Postlab 2	
Instructor's comments:				

1. Introduction / Theory of Operation

Throughout this lab, I will learn about Quartus, VCS, and continue my knowledge of Verilog HDL. This includes a review on continuous assignments, primitives, conditional operators, delays, Verilog value types, number representations, vectors, and Verilog operators.

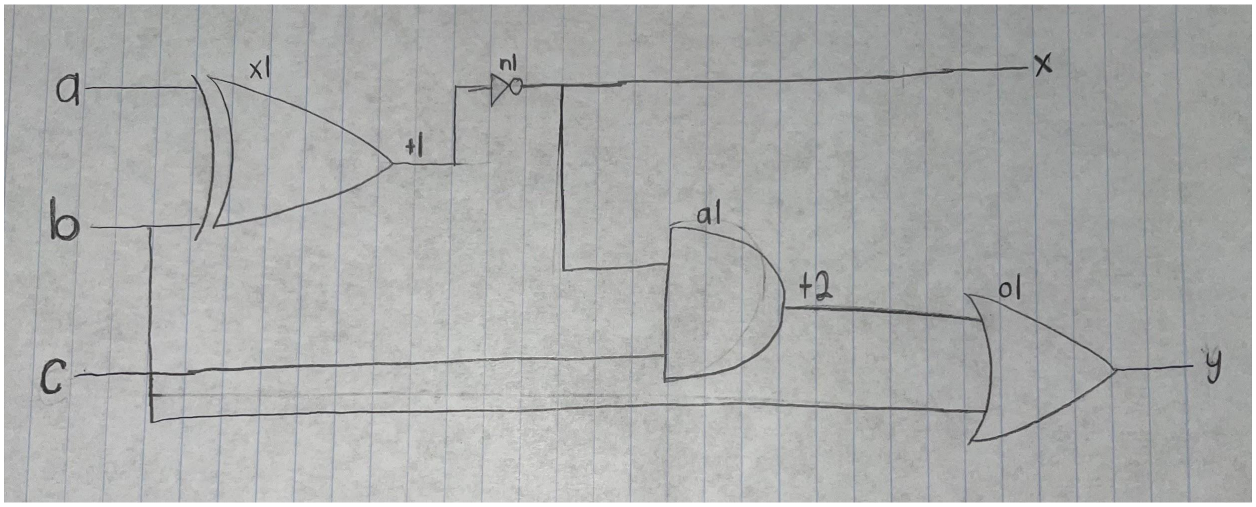
a. **Primitive**, in Verilog, is using gate operations in a more simple term by naming the specific gate and having the outputs on the left of the port list and inputs on the right of the port list. This makes instantiating easier as the built in component library would already assign the inputs to the output values rather than using the "assign" statement. One of the reasons why primitives could be used is to shorten a module as assigning continuous assignments with gates would sometimes be repetitive.

b. **Structural model**, in Verilog, is the assembly of modules that model basic components like digital gates and adders. A structural model would also introduce the hierarchy of how the inputs and outputs get carried out throughout various gates. Structural models are also digital schematic representations that allow the user to see how the inputs and outputs are propagated from the Verilog code.

c. **Dataflow**, in verilog, describes the flow of information (inputs and outputs). This is used for combinational circuits and gates as users would see how their Verilog code corresponds to various logical combinations. Dataflow could be viewed and interpreted through testbenches, waveforms, etc. This leads continuous assignments to be modelled through continuous signal flows from input changes to output changes.

2. Prelab

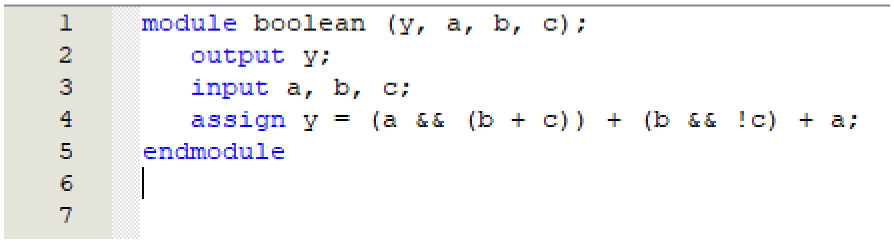
1.



This is my schematic/drawing for “module f1” showing all signals and wires.

2.

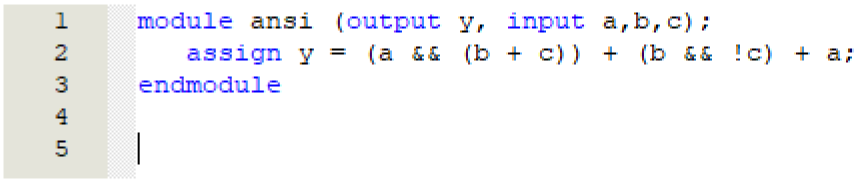
a.



```
module boolean (y, a, b, c);
   output y;
   input a, b, c;
   assign y = (a && (b + c)) + (b && !c) + a;
endmodule
```

This is my Verilog code for the boolean expression: “a(b+c) + bc’ + a” in normal Verilog style.

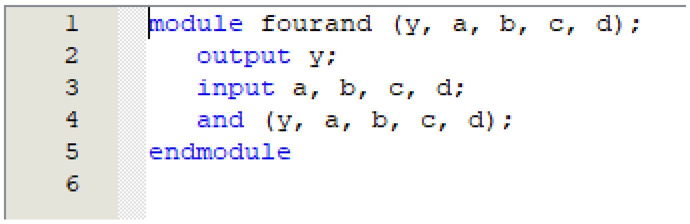
b.



```
module ansi (output y, input a,b,c);
   assign y = (a && (b + c)) + (b && !c) + a;
endmodule
```

This is my Verilog code for the boolean expression: “a(b+c) + bc’ + a” in ANSI-C style.

3.



```
module fourand (y, a, b, c, d);
   output y;
   input a, b, c, d;
   and (y, a, b, c, d);
endmodule
```

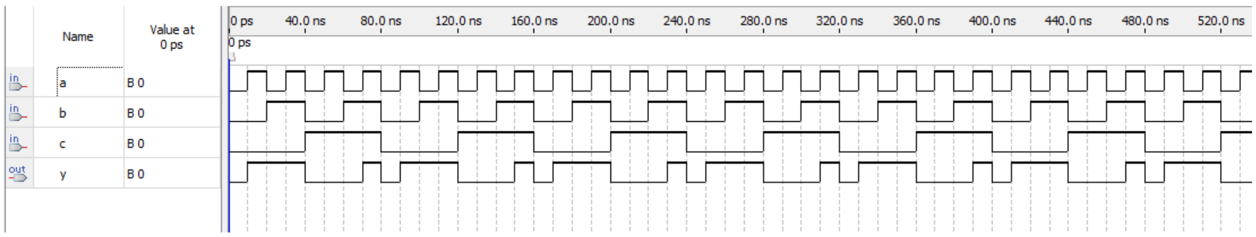
This is my Verilog code for a four input AND gate using gate primitive.

3. Results of Experiments

<u>Experiment 1</u>

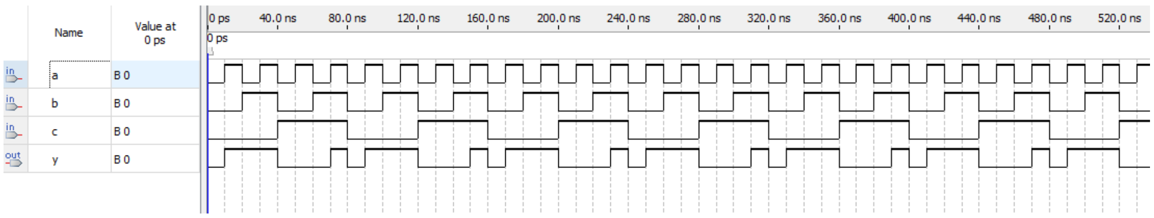
a.

a.



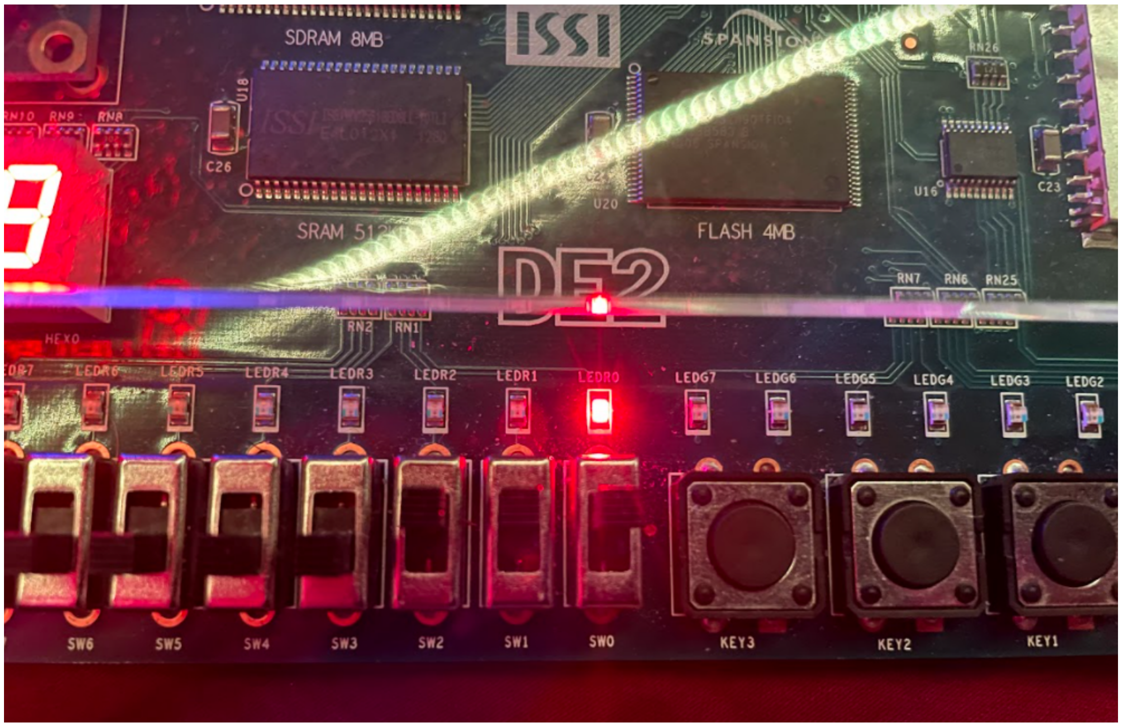
These are my waveforms for the Normal Verilog style of the Boolean expression: a(b+c)+bc'+a .

b.

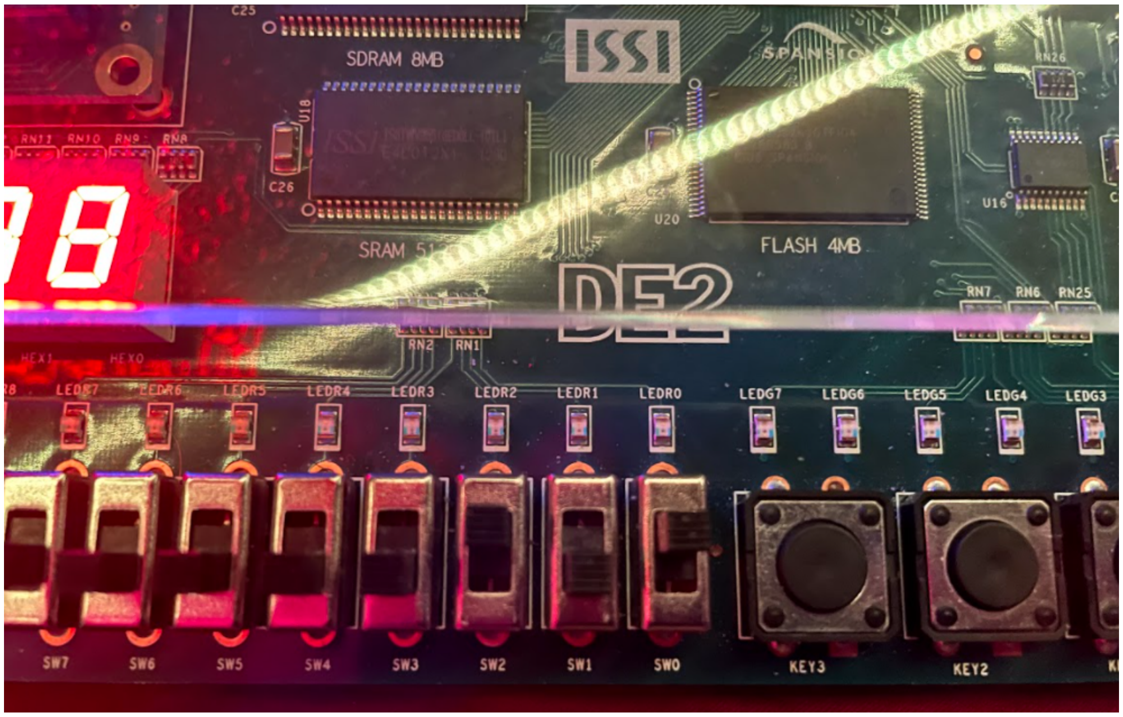


These are my waveforms for the ANSI-C style of the Boolean expression: a(b+c)+bc'+a . Ideally, it's the same waveform as Normal Verilog style.

b.



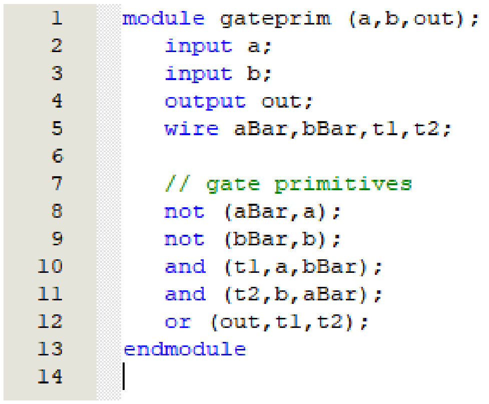
This is my DE2 board displaying the output of the ANSI-C style Boolean expression: a(b+c)+bc'+a . The output is 1 when a,b, and c are all 1.



This is my DE2 board displaying the output of the ANSI-C style Boolean expression: a(b+c)+bc'+a . The output is 0 when a and c are 1 but b is 0.

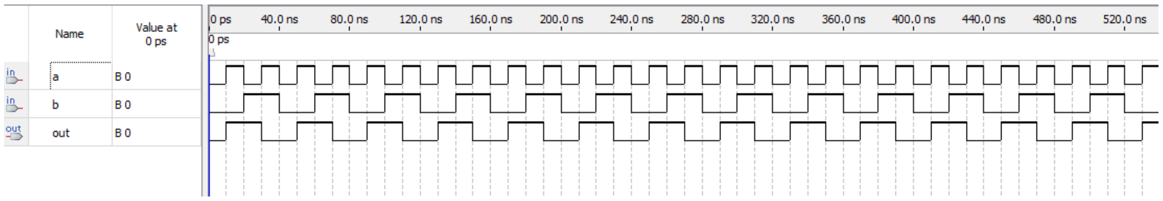
Experiment 2

a.



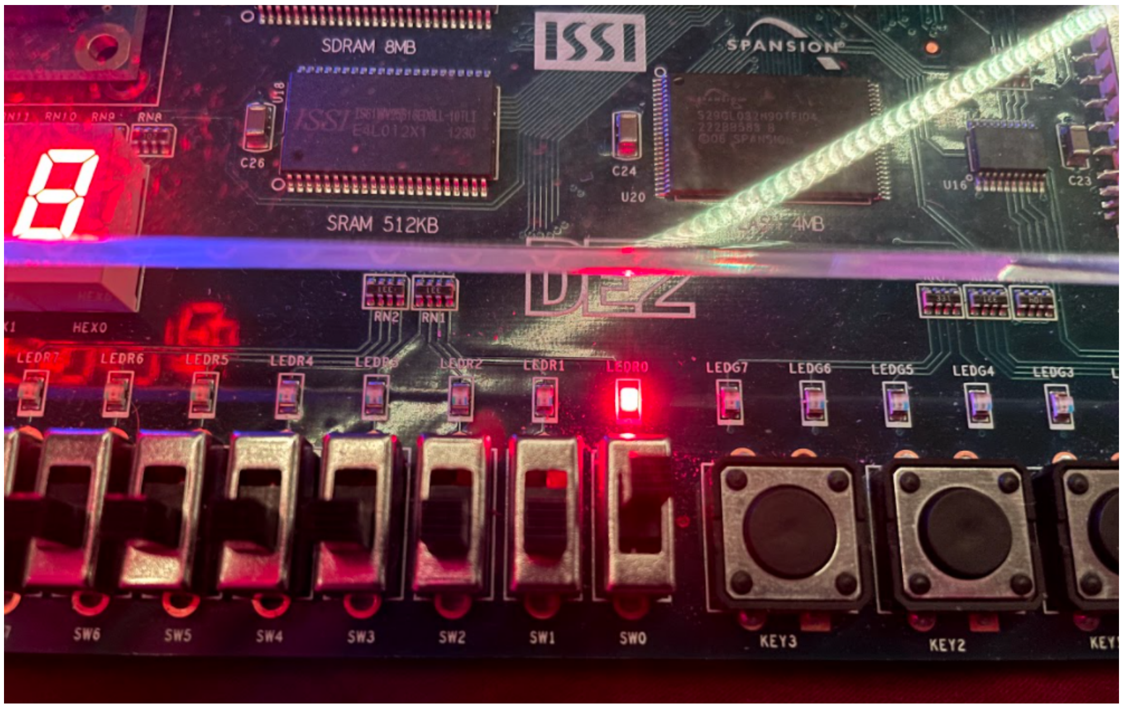
```
module gateprim (a,b,out);
   input a;
   input b;
   output out;
   wire aBar,bBar,t1,t2;

   // gate primitives
   not (aBar,a);
   not (bBar,b);
   and (t1,a,bBar);
   and (t2,b,aBar);
   or (out,t1,t2);
endmodule
```

This is my Verilog code for the provided schematic using gate primitives.

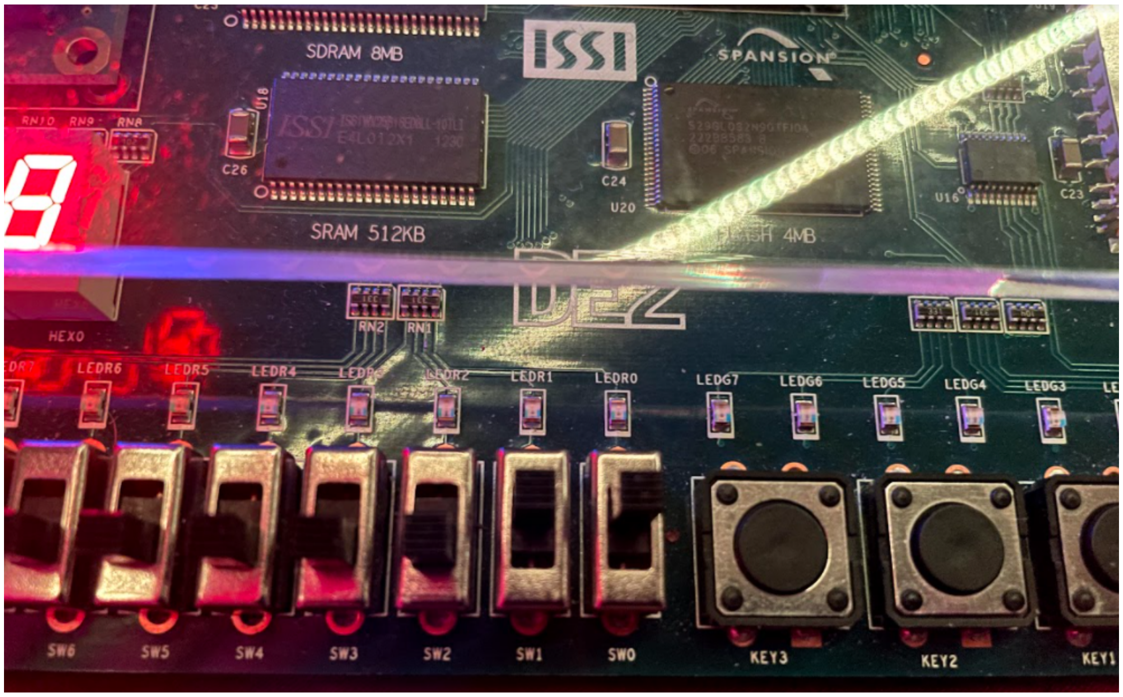


These are my waveforms for the gate primitive Verilog code.

b.



This is my DE2 board displaying the output of the gate primitive Verilog code . The output is 1 when a is 1 and b is 0.

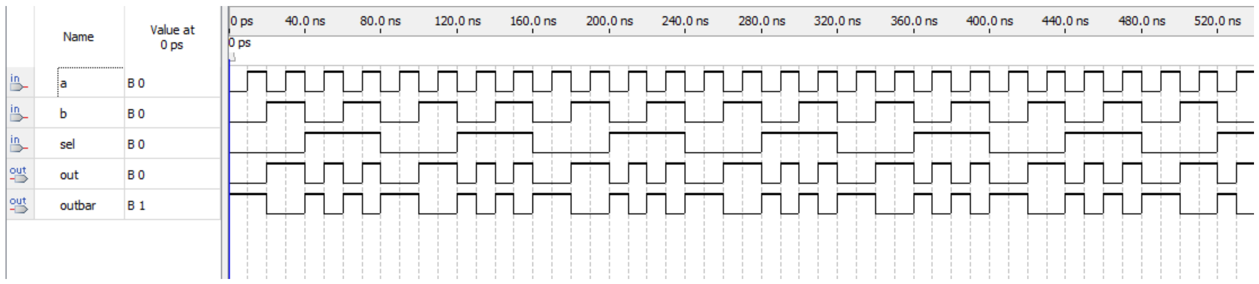


This is my DE2 board displaying the output of the gate primitive Verilog code . The output is 0 when a is 1 and b is 1.

Experiment 3

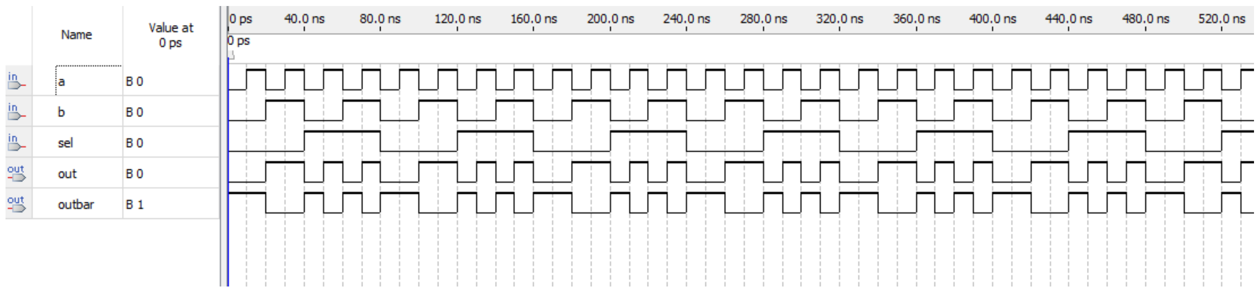
a.

a.



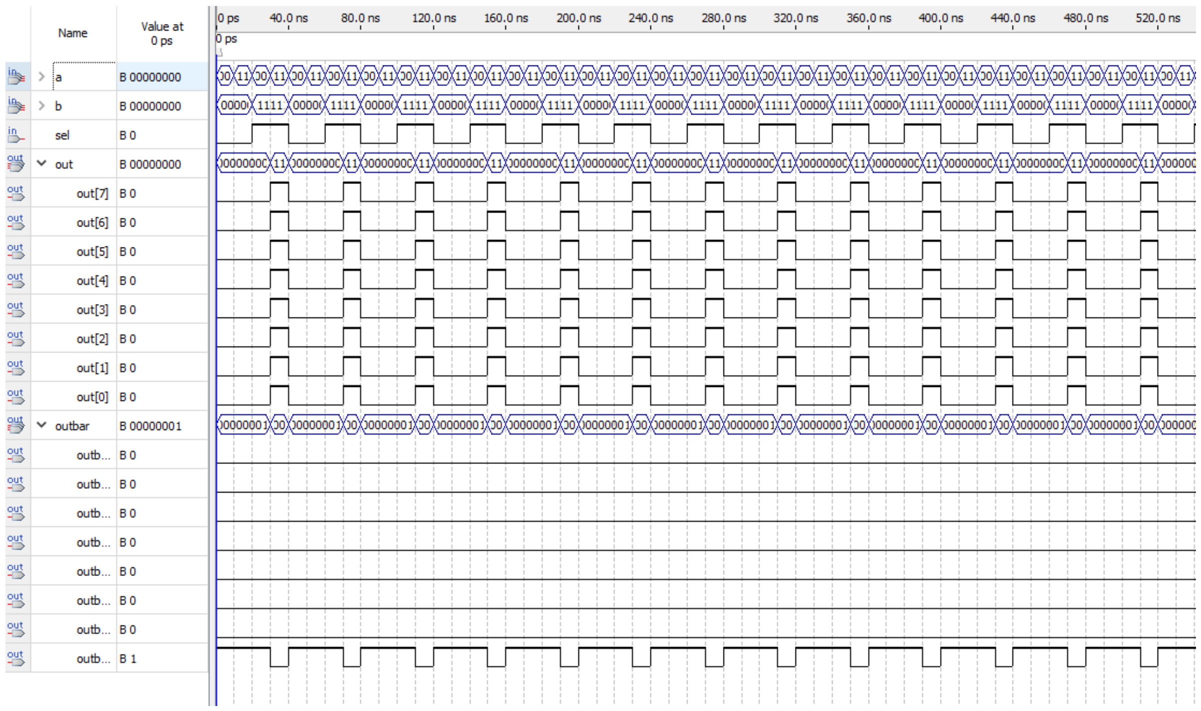
These are my waveforms for a 2-to-1 MUX using continuous assignments.

b.



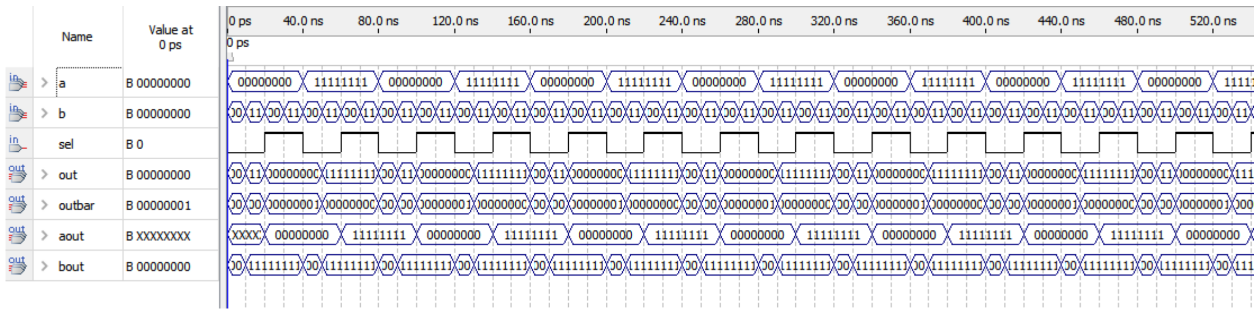
These are my waveforms for a 2-to-1 MUX using the conditional operator.

c.



These are my waveforms for a 2-to-1 MUX with 8-bit inputs and 8-bit outputs.

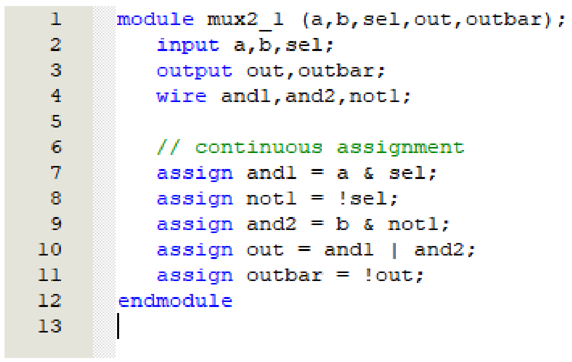
d.


Name	Value at 0 ps
a	B 00000000
b	B 00000000
sel	B 0
out	B 00000000
outbar	B 00000001
aout	B XXXXXXXX
bout	B 00000000

These are my waveforms for a 1-to-2 DEMUX using the output of the 2-to-1 MUX.

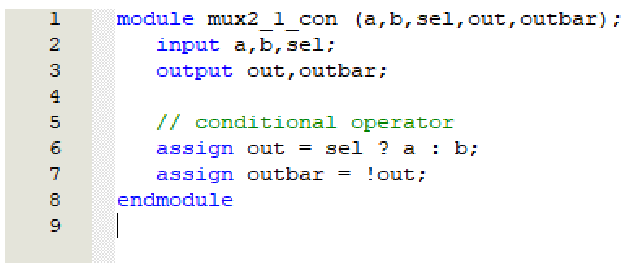
b.

a.


```
module mux2_1 (a,b,sel,out,outbar);
   input a,b,sel;
   output out,outbar;
   wire and1,and2,not1;

   // continuous assignment
   assign and1 = a & sel;
   assign not1 = !sel;
   assign and2 = b & not1;
   assign out = and1 | and2;
   assign outbar = !out;
endmodule
```

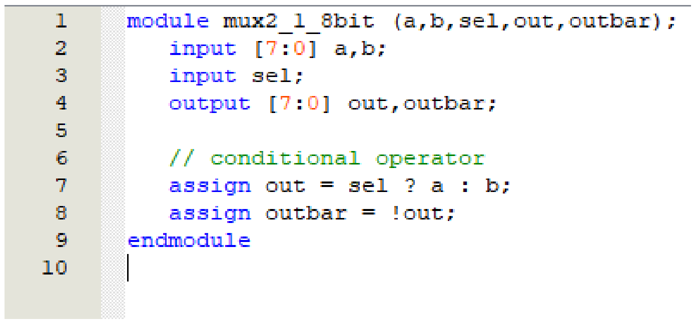
This is my Verilog code for the 2-to-1 MUX using continuous assignments.

b.


```
module mux2_1_con (a,b,sel,out,outbar);
   input a,b,sel;
   output out,outbar;

   // conditional operator
   assign out = sel ? a : b;
   assign outbar = !out;
endmodule
```

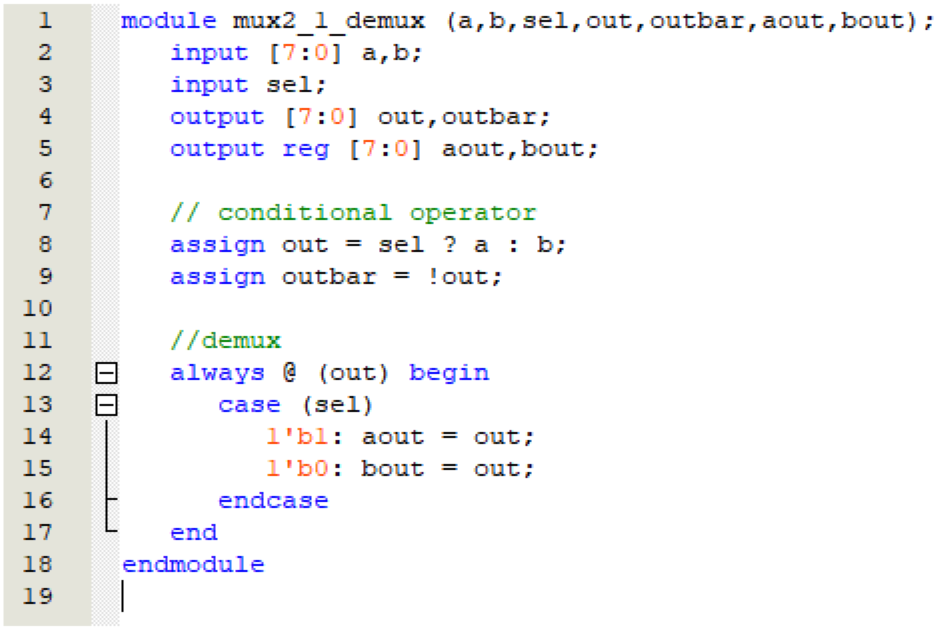
This is my Verilog code for the 2-to-1 MUX using the conditional operator.

c.


```
module mux2_1_8bit (a,b,sel,out,outbar);
   input [7:0] a,b;
   input sel;
   output [7:0] out,outbar;

   // conditional operator
   assign out = sel ? a : b;
   assign outbar = !out;
endmodule
```

This is my Verilog code for the 2-to-1 MUX with 8-bit inputs and 8-bit outputs.

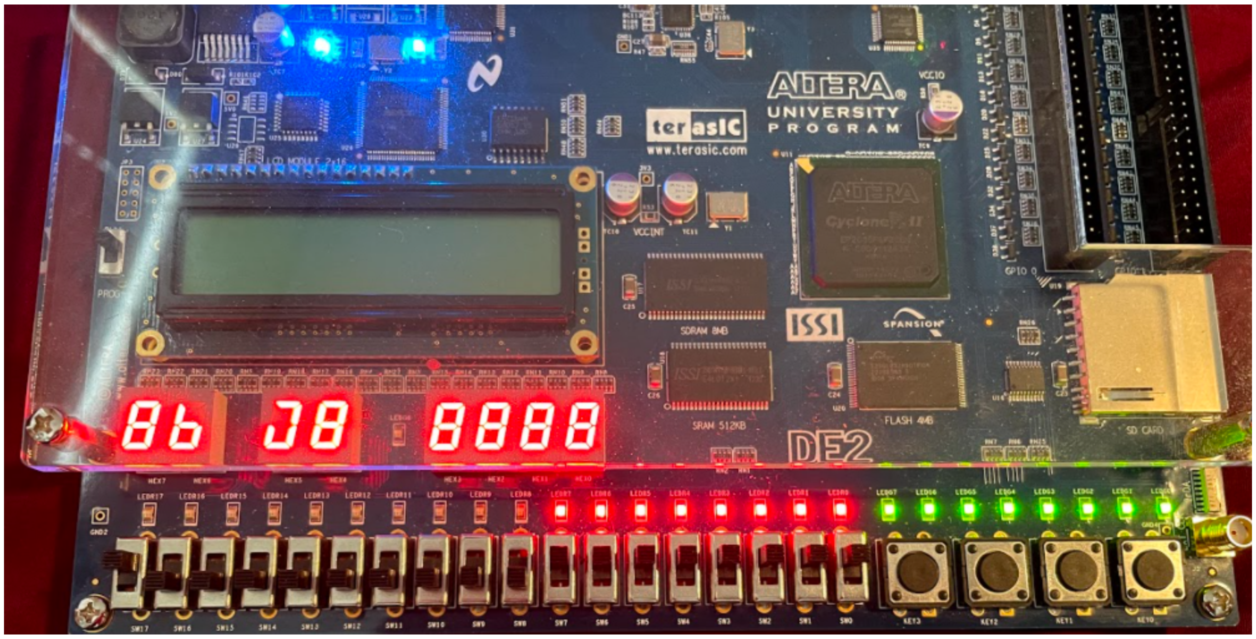
d.



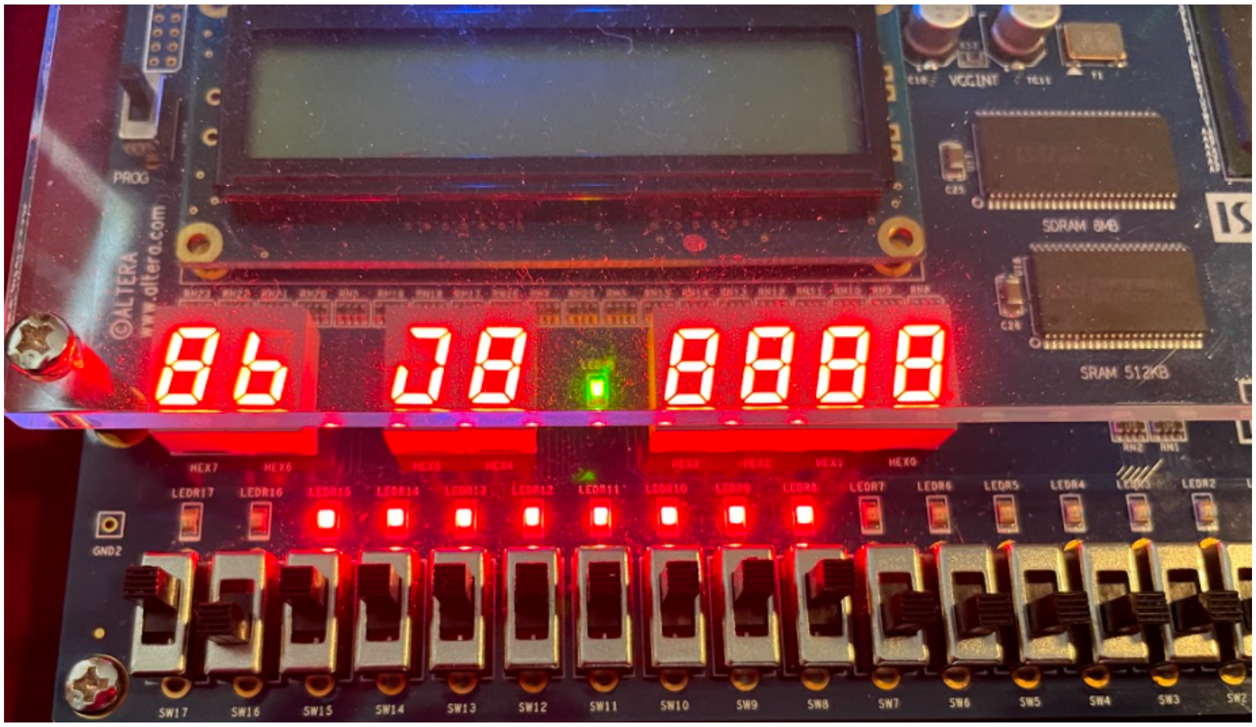
```
module mux2_1_demux (a,b,sel,out,outbar,aout,bout);
   input [7:0] a,b;
   input sel;
   output [7:0] out,outbar;
   output reg [7:0] aout,bout;

   // conditional operator
   assign out = sel ? a : b;
   assign outbar = !out;

   //demux
   always @ (out) begin
      case (sel)
         1'b1: aout = out;
         1'b0: bout = out;
      endcase
   end
endmodule
```

This is my Verilog code for the 1-to-2 DEMUX using the output of the 2-to-1 MUX.

c.



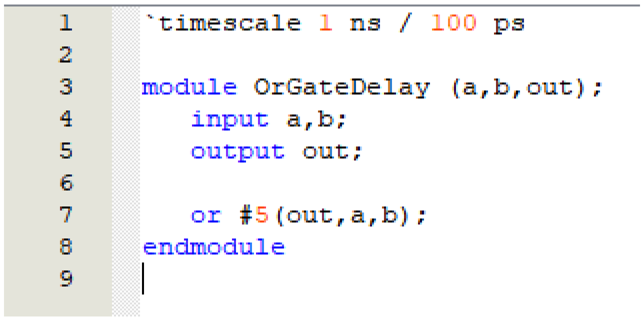
This is my DE2 board displaying the output of the 1-to-2 DEMUX using the output of the 2-to-1 MUX . The output for the 2-to-1 MUX is displayed with the green LEDS; therefore, when all of the a-inputs are 1 and when sel is 1. The output for the 1-to-2 DEMUX is displayed with red LEDS; therefore, the output is the same as the input to the 2-to-1 MUX.



This is my DE2 board displaying the output of the 1-to-2 DEMUX using the output of the 2-to-1 MUX . The output (outbar) for the 2-to-1 MUX is displayed with the 1 green LED; therefore, when all of the b-inputs are 1 and when sel is 1. The output for the 1-to-2 DEMUX is displayed with red LEDS; therefore, the output is the same as the input to the 2-to-1 MUX.

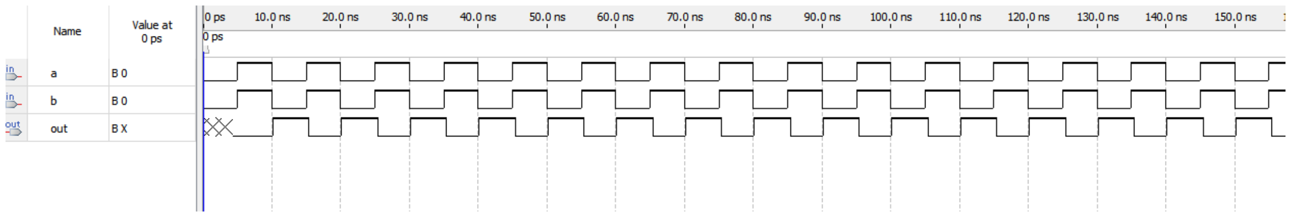
Experiment 4

a.



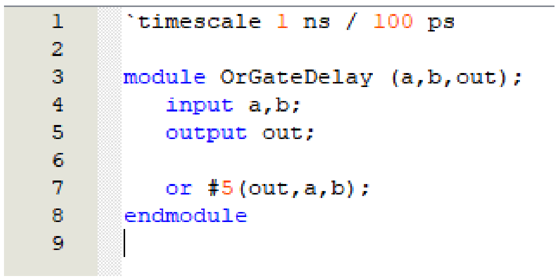
```
`timescale 1 ns / 100 ps

module OrGateDelay (a,b,out);
   input a,b;
   output out;

   or #5(out,a,b);
endmodule

```

This is my Verilog code modeling the OR gate having the 5ns gate delay.



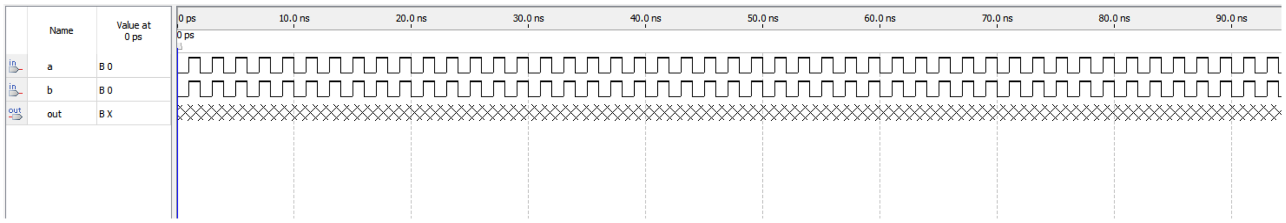
These are my waveforms for the simulation of the OR gate for the gate input state changing every 10 ns.

b.



```
`timescale 1 ns / 100 ps

module OrGateDelay (a,b,out);
   input a,b;
   output out;

   or #5(out,a,b);
endmodule
```

This is my Verilog code modeling the OR gate having the 5ns gate delay.

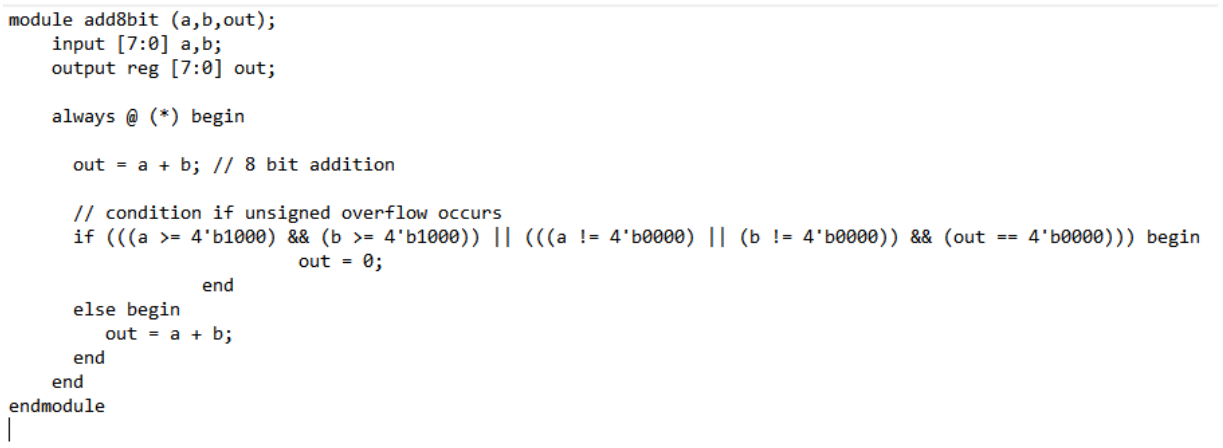


These are my waveforms for the simulation of the OR gate for the gate input state changing every 2 ns.

c. The difference between the 10 ns delay and the 2 ns delay is quite clear. When the input changes for every 10ns, the output is simulated with the 5 ns delay; therefore, the waveforms show that the output is 1 when a and b are 0. Here, the output is revealed to have the delay at the beginning of the output wave simulation. In contrast, the 2ns delay reveals the output to consist of only X (the simulator can't decide the value). There is no output and there is no 5ns delay shown here. The reason behind this is because the input signal changes more often than the 5ns as the input state changes for every 2 ns. In conclusion, the OR gate having the 5ns gate delay would only show the output if the input state changes for 5ns or more.

Experiment 5

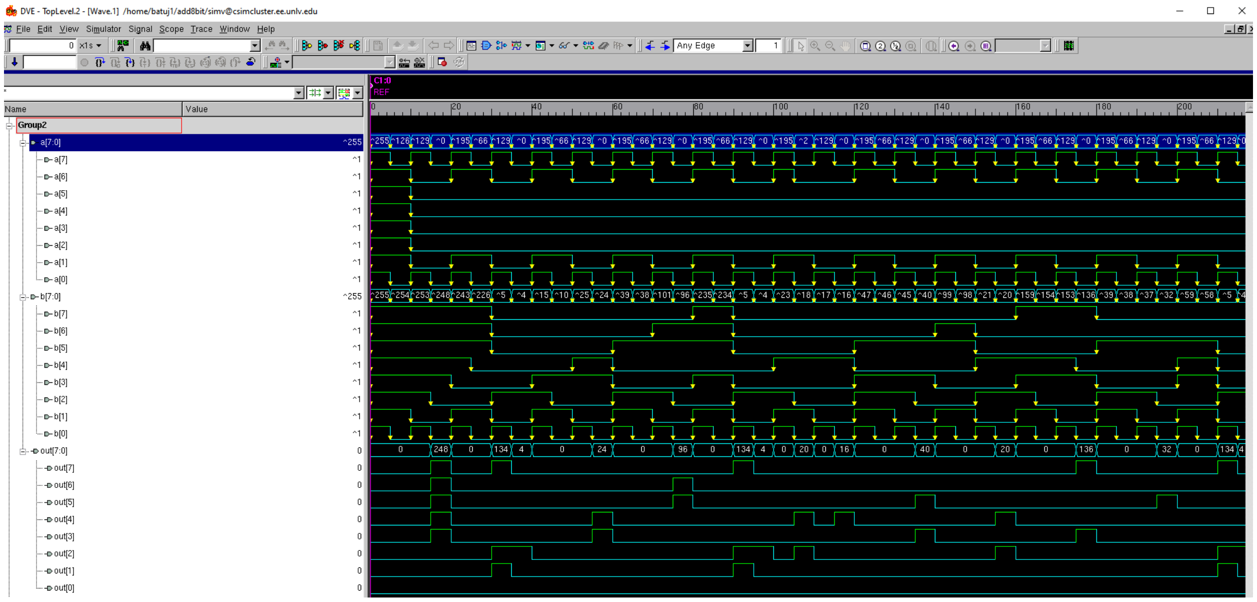
a.



```
module add8bit (a,b,out);
    input [7:0] a,b;
    output reg [7:0] out;

    always @ (*) begin

      out = a + b; // 8 bit addition

      // condition if unsigned overflow occurs
      if (((a >= 4'b1000) && (b >= 4'b1000)) || (((a != 4'b0000) || (b != 4'b0000)) && (out == 4'b0000))) begin
                                 out = 0;
                        end
      else begin
         out = a + b;
      end
    end
endmodule
```

This is my Verilog code for implementing 8-bit addition. There is also an if statement that also includes a situation where there is unsigned overflow.

b.


DVE - TopLevel.2 - [Wave.1] /home/batuj1/add8bit/simv@csimcluster.ee.unlv.edu
File Edit View Simulator Signal Scope Trace Window Help
Any Edge
Name
Value
Group2
a[7:0]
b[7:0]
out[7:0]

This is my VCS waveform for the 8-bit addition. Included in this screenshot are situations where output would be 0 where unsigned overflow occurs.

4. Answers to questions

Question 1: What is the Verilog preprocessor directive?

The Verilog preprocessor directive is a compile instruction used to run the compilation of a Verilog code. Some examples of a Verilog preprocessor directive include: \`include, \`timescale, \`define, etc. Preprocessor directive will always include a " \` " mark before the directive instruction. The directive is effective from when it is stated to when another directive overrides it. They can be put anywhere within the module; however, it is common to see these directives utilized outside the module statement.

Question 2: What is the difference between == and === operators?

The == operator is a logical equality operator, and the === operator is a logical case equality operator. The main difference between the two is that the == operator is used to test for 1s and 0s, otherwise being a result of x; the === operator tests for 1s, 0s, z's, and x's. For example, a == b would compare a being equal to b with the result being unknown. In comparison, a === b would compare a equal to b as well as x and z.

Question 3: Is Verilog case sensitive? If yes, what does it mean?

Verilog HDL is case-sensitive. This means that lowercase and uppercase letters would dictate which keywords or identifiers are to be used within the module. Keywords, Quartus II primitive names, gate primitives, etc. must all be lowercase; thuse, a syntax error would occur if they were to be capitalized. For example, the keyword "module" must only be lowercase. In addition, constants would be capitalized as such: "parameter EXAMPLE = 0;". The "EXAMPLE" is indicated as a constant and the "parameter" would be a keyword.

Question 4: List types of nets in Verilog.

Nets (networks) represent structural connections between hardware entities. These nets do not hold any values; thus, they can constantly alter throughout the circuit. Some types of nets include: wire, wor, trior, tri, wand, triand, trireg, etc.

wire / tri: connects input and output ports of a module instantiation to another element

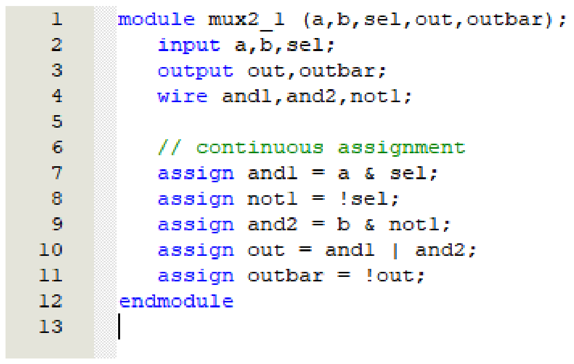
wor / trior: depends on logical "or" of the drivers connected to it

wand / triand: depends on logical "and" of the drivers connected to it

trireg: retains the last value when driven by a tristate

Question 5: Label the structural and behavioral model in experiment 3.

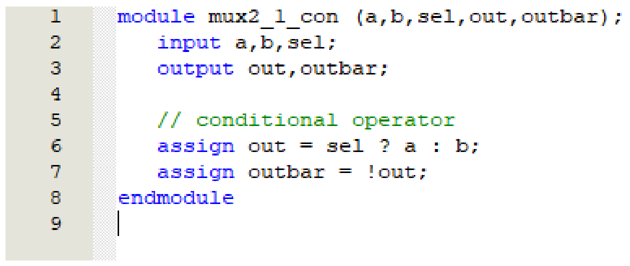
a.



```
module mux2_1 (a,b,sel,out,outbar);
   input a,b,sel;
   output out,outbar;
   wire and1,and2,not1;

   // continuous assignment
   assign and1 = a & sel;
   assign not1 = !sel;
   assign and2 = b & not1;
   assign out = and1 | and2;
   assign outbar = !out;
endmodule
```

The 2-to-1 MUX using continuous assignments is a behavioral model (describes the output as a function of inputs).

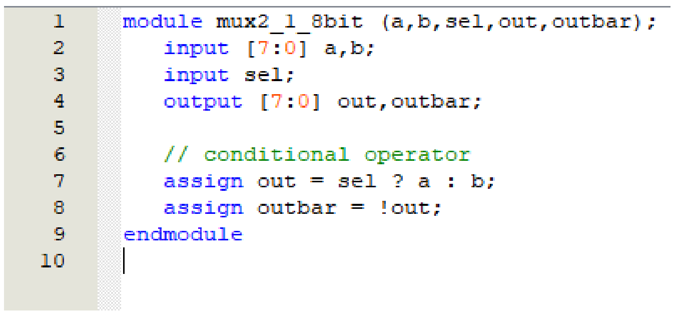
b.



```
module mux2_1_con (a,b,sel,out,outbar);
   input a,b,sel;
   output out,outbar;

   // conditional operator
   assign out = sel ? a : b;
   assign outbar = !out;
endmodule
```

The 2-to-1 MUX using the conditional operator is a behavioral model (describes the output as a function of inputs).

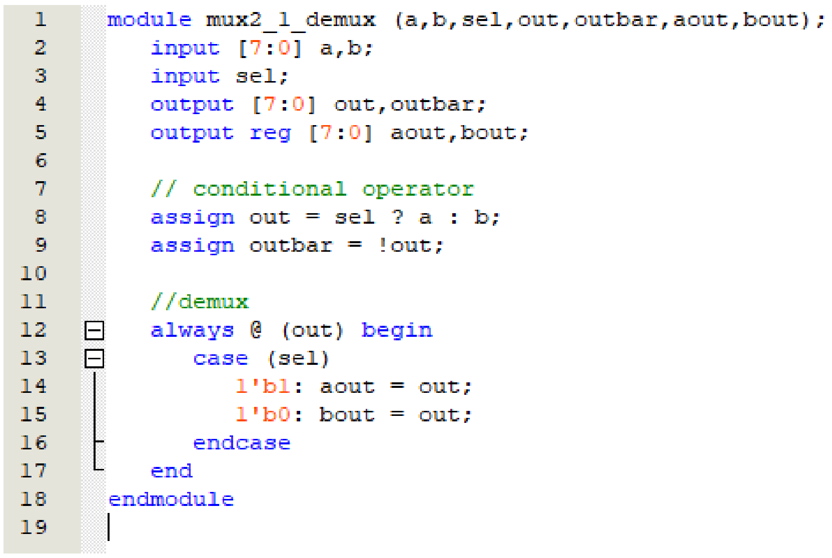
c.



```verilog
module mux2_1_8bit (a,b,sel,out,outbar);
   input [7:0] a,b;
   input sel;
   output [7:0] out,outbar;

   // conditional operator
   assign out = sel ? a : b;
   assign outbar = !out;
endmodule
```

The 8-bit 2-to-1 MUX is a behavioral model (describes the output as a function of inputs).

d.



```verilog
1  module mux2_1_demux (a,b,sel,out,outbar,aout,bout);
2     input [7:0] a,b;
3     input sel;
4     output [7:0] out,outbar;
5     output reg [7:0] aout,bout;
6
7     // conditional operator
8     assign out = sel ? a : b;
9     assign outbar = !out;
10
11    //demux
12    always @ (out) begin
13       case (sel)
14          1'b1: aout = out;
15          1'b0: bout = out;
16       endcase
17    end
18 endmodule
19
```

The 8-bit 1-to-2 DEMUX connected to an 8-bit 2-to-1 MUX consists of only behavioral models as lines 8 to 9 describe the output as function of inputs, and lines 12 to 17 describe the function of a circuit by manipulating the variables of the input data types.

Since a structural model describes a module composed of built-in gate primitives and simpler modules, none of experiment 3 is structural.

5. Conclusions & Summary

This lab was pretty easy until Experiment 3 part d. I solved Experiment 3 part d, by doing some research on how a demux was built as I've never used a demux before. By knowing that the demux only takes one input and outputs two values, I came to the conclusion that the inputs of the mux should match the outputs of the demux. In addition, I came across some problems regarding the delay as I've never done delay within a module without using a testbench. The only time I've used delay was to create testbenches; therefore, I solved this issue by watching the Lab 2 video and asking the TA for assistance. This lab was quite a challenge compared to the first lab; however, I did learn a lot of new tools that would help me throughout the lab in the future.7