LAB #3: ADDERS and COMPARATORS using 3 types of Verilog Modeling

LAB OBJECTIVES

1. Practice designing more combinational logic circuits
2. More experience with equations and the use of K-maps and Boolean Algebra
3. Practice designing combinational logic circuits with NAND gates
4. More use of the Xilinx programmable logic device (PLD)
5. More advance uses of Verilog (Data Flow and Behavioral)
6. More exposure to using the ISE tool

LAB PROCEDURE

PART 1 - Design (create truth table, use K-maps to find equations, and create a circuit schematic diagram) for: A four-bit adder that accepts any 4-bit input and adds a 4-bit binary constant to the input. The output needs five bits. For example, if the input was 0111 and the constant is 0101 the output should be 01100. (Note: Your instructor will assign the constant to you in lecture) **Y**ou should have five equations for the five outputs of the adder. Call them sum4, sum3, sum2, sum1, and sum0. Using only one package of: 74LS00 {NAND}, one 74LS04 {NOT}, one 74LS08 {AND} and one 74LS32 {OR} (and your creativity), design a circuit for sum0, sum1, sum2, sum 3 and sum4, however, by this time you may be running out of available gates. Remember to use DeMorgan's theorems and sharing of circuits between outputs. Remember there are 4 gates available in each package of the NAND, AND, and OR Integrated Circuit (IC), and 6 gates in the NOT IC. That's a total of 18 gates that are available for you to use! Wire your circuit carefully. You do not want to make a mistake. If you do make a wiring error, it will take time to find it. We call this process "Debugging". You might call it "frustration". Test your circuit by changing the inputs. Your output should ADD your constant to the four inputs. Record the outputs in a truth table for your lab report. There are 16 possible combinations of inputs. Also include a "complete" schematic with Reference Designators (U#'s) and with the IC pin #s. (1,3,5,7,9,B,D,F)

PART 2 - Use your **five** equations for the ADDER circuit from Lab2 part 1 and write a **Data Flow** model description in Verilog HDL. **Be sure you have 5 equations**. If you did not do 5 equations in part 1, then you need to derive them for this part. You should write the Verilog at home using a TEXT EDITOR (e.g. notepad or Wordpad). (If you have ISE at home, you can develop the entire design). Bring *.v file to Lab class and compile and download your Verilog solution to the Spartan3E board. Test your adder for all possible combinations. If your adder is "doing something", but not working properly, you will need to check your truth table, equations, your K-mapping, any Boolean Algebra and DeMorgan's that you used, and the Verilog file, your ISE procedure and possibly the Spartan3E board too. It's important to perform your troubleshooting techniques efficiently and methodically.

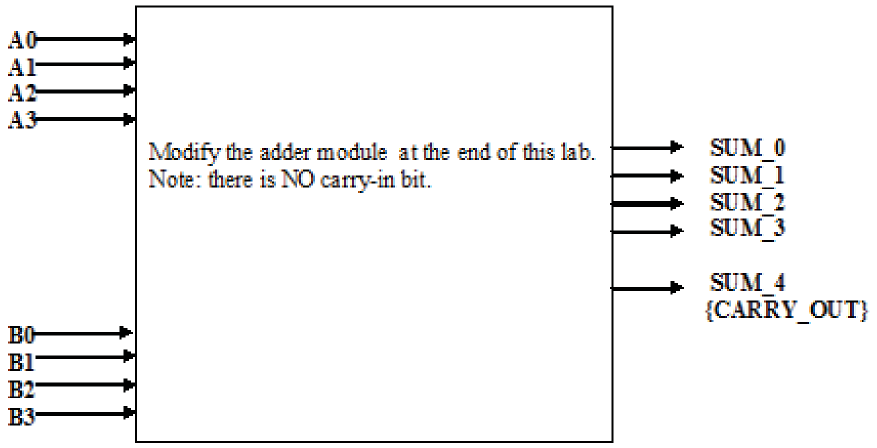
FAQ - What does "doing something" mean? Good question! If your outputs are stuck low, or stuck high, or some are stuck low and some are stuck high, then your PLD is not doing anything. If the outputs change in some mysterious fashion, but not correctly as you change inputs, we say the PLD is at least doing something. When the outputs of a PLD appear to be "doing something", you can assume that the PLD has at least been programmed (or what we called "downloaded"). Since PLDs have non-volatile memory that is they keep what was downloaded last, you may be observing what someone downloaded earlier in the day. Or if your code was downloaded successfully, you may have problems with your equations. Many times, miss assigning pin numbers for switches and LED's causes problems as well.

PART 3 - Design a full 4-bit adder that adds any 4 bits to any 4 bits. The two 4-bit numbers shall be interpreted as **unsigned**. Call the 2 numbers **A** and **B**. The Function will be **A plus B.**
The number **A** has four bits called **a3, a2, a1** and **a0**. The number **B** has four bits called **b3, b2, b1 and b0.** The adder shall have five outputs called S4 {or **CO** for Carry Out}, **S3** {Sum 3}, **S2** {Sum 2}, **S1** {Sum 1}, **S0** {Sum 0}.

Don't do a truth table, but in your lab report discuss how many rows there would be in the truth table if you had to do one. Design tools such as Verilog will "design" functional units like adders for you! Examine the Verilog addition example at the end of this lab and modify it to make it a full 4-bit adder. The Verilog used in the addition example is done using “Behavior” modeling. You should use the same style. Compile, download, and test your design on the Spartan3E board.

Connect your two sets of 4 inputs to switches (total of 8 switches). Connect the 5 Sum outputs to LEDs.

Ask your instructor to tell you which specific switches and LEDs for you to assign your I/O to.


A0
A1
A2
A3
Modify the adder module at the end of this lab.
Note: there is NO carry-in bit.
SUM_0
SUM_1
SUM_2
SUM_3
SUM_4
{CARRY_OUT}
B0
B1
B2
B3

Off-Line - Maybe you noticed schematics while paging through the textbook, or other logic design texts. Many, many years ago (20 years ago is ancient history for computer engineering) designers did all their work using schematics. They moved from drafting tables to drafting software (heard the name ORCAD, AUTOCAD, etc.?) about 15 years ago. Today very few companies use schematics, even computer-generated schematics, for any designs similar to this lab. Everyone uses programming languages such as Verilog. However, schematic capture still is important. For one thing many older design engineers still think in terms of schematics instead of equations (this includes the authors of logic design textbooks). Electronic technicians probably like schematics better than equations. Large schematics printed on color printers make great apartment wall coverings. And, seriously, schematics can be very helpful in high-level, advanced logic design where many modules are interconnected together.

PART 4 - Design of Comparator Using Gates:

Purpose:

To design a simple combinational circuit, i.e., 2-bit comparator, and to gain insight into IC chip minimization instead of gate minimization in implementing the combinational circuit.

Procedure:

Design a 4-input, 3-circuit that compared two 2-bit unsigned numbers. You can call these numbers a1 a0 and b1 b0. So here a1 is the most significant bit of input A, and a0 is the least significant bit of input A. This circuit should have 3 outputs, which indicate whether A > B, A = B or A < B. You can label these outputs G (which means a1 a0 > b1 b0, where G stands for greater), E (which means a1 a0 + b1 b0), and L (which means a1 a0 < b1 b0). This is called a comparator circuit, because it compares the binary number a1 a0 with the binary number b1 b0.

The outputs should be connected to LED indicator circuits. Note that an output of 1 means that the condition being tested by that output is true, which should turn the LED on, while an output of 0 means the condition being tested is false, which should turn the LED off.

Start this problem with a truth table that describes the problem. Try to minimize the number of integrated circuit packages needed. One way is with K-maps. You may be able to further minimize the number of gates by thinking about what these outputs really mean, in English. Think about the relationship between the three outputs. The minimize number of chips needed is about 5. You may get graded down for using more gates than needed.

Use only NAND gates (74LS00, 74LS10, 74 LS20). Draw a circuit diagram, making sure to include pin numbers, reference designators, and a parts list. Connect switches to your inputs. Connect the three outputs to the three LED indicators circuits from lab A on your own breadboard. Draw gates to look gates in circuit diagrams, not like boxes with pins. Construct and test the circuit. Demonstrate this circuit to your lab instructor.

```
module comparator (a0,a1,b0,b1,G,E,L);
input   a0,a1,b0,b1;
output  G,E,L;
reg     G,E,L;

always@(a0 or a1 or b0 or b1)
  begin
    L<={a1,a0}<{b1,b0};       // less than

     // greater than
     if ({a1,a0}>{b1,b0})
          G<= 1;
     else
          G<= 0;
     // equivalent to G={a1,a0}>{b1,b0};

E<={a1,a0}=={b1,b0};  //logical equality
   end
endmodule
```

PART 5 - Design an **unsigned** 3-bit comparator such that “A” is compared to “B”.

Inputs: A three bits (a2, a1, a0)	**Outputs:**	**E**	**A equal to B**
B three bits (b2, b1, b0)		**L**	**A less than B**
		G	**A greater than B**

E, L, and G are each one bit

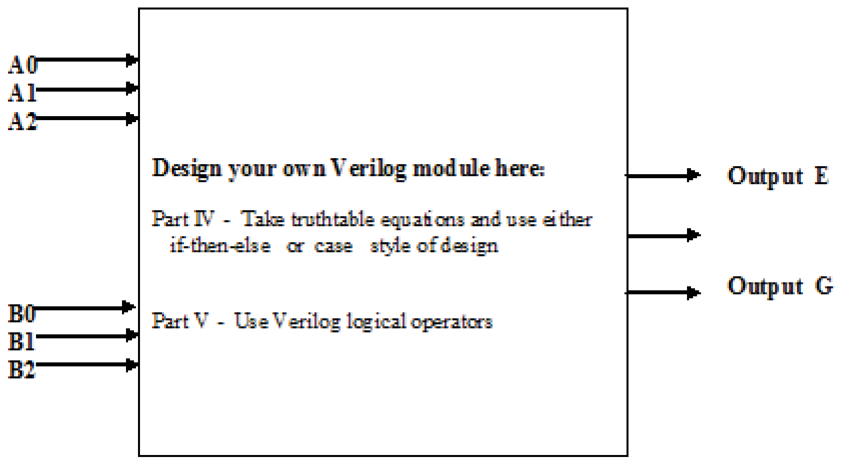
Example: if A = 010 and B = 100, then the outputs would be: E=1, L=0 and G=1 {if active low outputs}

Write this problem statement in the **form** of a truth table. There will be 6 columns for inputs and 3 columns for outputs. You could use K-maps to simplify the equations for E, L and G. How many variables would this K-map have? With Verilog you could skip the K-maps and design a solution based on a truth table and all the input possibilities. The "if-then-else if" Verilog construct would have 64 entries. A "case" construct would have 64 entries as well. Pick one or the other and design a Verilog solution. What kind of modeling technique is this called? With a good text editor you can copy, paste and edit changes rather quickly. Compile, download and test your "truth table" type of solution. Include your truth table, your Verilog source listing, and the segment of the report file that included the equations actually built by ISE in your lab report.

Instead of doing the above **unsigned** comparator, design your comparator to “interpret” the A and B numbers as “SIGNED” in twos complement format.

PART 6 - While the solution for Part 4 was better than doing a six variable K-map, it still was a lengthy design solution (lots of lines of Verilog code). You would suspect that Verilog is better than this! Verilog has the following "logical" operators: = = equality, > greater than, < less than, >= greater than or equal, =< less than or equal to, and != inequality. Be adventuresome and redesign a solution to the comparator problem by using the relational operators. Compile, download, and test your design. Again, include the segment of the report file that included the equations actually built by ISE using this style of Verilog. Compare the equations with part 4 and comment about any differences in the equations using the two techniques done in each part.

Add a signal called U/S (Unsigned/Signed). When the signal is **High** (Unsigned mode), the Comparator interprets the numbers as Unsigned numbers. When the signal is **Low** (Signed Mode), the Comparator interprets the numbers as signed numbers.


A0
A1
A2
B0
B1
B2
Design your own Verilog module here:
Part IV - Take truthtable equations and use either if-then-else or case style of design
Part V - Use Verilog logical operators
Output E
Output G

EXAMPLES of Verilog HDL

```
module simple_2 (a,b,c,d,f1,f2,f3);
input a,b,c,d;
output f1,f2,f3;
assign f1 = (a & ~b) | (~a & b); // f1 = a xor b
assign f2 = (c & ~d) | (~c & d); // f2 = c xor d
assign f3 = (f1 & ~f2) | (~f1 & f2); // f3 = f1 xor f2
endmodule
```

A two-bit adder:

```
module adder(a1,a0,b1,b0,c_in,c_out,s1,s0);
input a1,a0,b1,b0,c_in;
output c_out,s1,s0;
assign {c_out,s1,s0}={a1,a0}+{b1,b0}+c_in;
endmodule
```

Example of Verilog CASE statement use:

```
module case_1 (a,b,c,f1,f2);
input a,b,c;
output f1,f2;
reg f1,f2;
always@( a or b or c )
begin
case ({a,b,c})
0 : begin
f1<=0; f2<=0;
end
```

```
1 : begin
f1<=0; f2<=1;
end
2 : begin
f1<=0; f2<=1;
end
3 : begin
f1<=0; f2<=0;
end
4 : begin
f1<=1; f2<=0;
end
5 : begin
f1<=1; f2<=1;
end
6 : begin
f1<=1; f2<=1;
end
7 : begin
f1<=1; f2<=0;
end
default: begin
f1<=0; f2<=0;
end
endcase
end
endmodule
```

Example of Verilog IF, ELSE IF statement use:

```
module ifthen_1 (a,b,c,f1,f2);

input   a,b,c;
output f1,f2;

reg      f1,f2;

always@(a or b or c )
 begin
  if (a==0 && b==0 && c==0 )
    begin
          f1<=0; f2<=0;
    end
  else if (a==0 && b==0 && c==1 )
    begin
          f1<=0; f2<=1;
    end
  else if (a==0 && b==1 && c==0 )
    begin
          f1<=0; f2<=1;
   end
  else if (a==0 && b==1 && c==1 )
    begin
          f1<=0; f2<=0;
    end
  else if (a==1 && b==0 && c==0 )
    begin
          f1<=1; f2<=0;
    end
  else if (a==1 && b==0 && c==1 )
    begin
```

```
          f1<=1; f2<=1;
   end
  else if (a==1 && b==1 && c==0)
   begin
          f1<=1; f2<=1;
   end
  else if (a==1 && b==1 && c==1 )
   begin
          f1<=1; f2<=0;
   end

end
endmodule
```

```
module
adder(a0,a1,b0,b1,c_in,s0,s1,c_out);

input   a0,a1,b0,b1,c_in;
output  s0,s1,c_out;

assign
{c_out,s1,s0}={a1,a0}+{b1,b0}+c_in;

endmodule
```

```
module ifthen_2 (a,f);

input  [3:0] a;
output [2:0] f;

reg [2:0] f;

always@(a)
 begin
         if (a==0)        f<=3'b000;
         else if (a==1)   f<=3'b011;
         else if (a==2)   f<=3'b011;
         else if (a==3)   f<=3'b000;
         else if (a==4)   f<=3'b101;
         else if (a==5)   f<=3'b110;
         else if (a==6)   f<=3'b110;
         else if (a==7)   f<=3'b101;
         else if (a==8)   f<=3'b101;
         else if (a==9)   f<=3'b110;
         else if (a==10)  f<=3'b110;
         else if (a==11)  f<=3'b101;
         else if (a==12)  f<=3'b000;
         else if (a==13)  f<=3'b011;
         else if (a==14)  f<=3'b011;
         else             f<=3'b000;

 end

endmodule
```

```
module comparator (a0,a1,b0,b1,G,E,L);

input   a0,a1,b0,b1;
output  G,E,L;

reg     G,E,L;

always@(a0 or a1 or b0 or b1)
  begin
     L<={a1,a0}<{b1,b0};        // less than

      // greater than
      if ({a1,a0}>{b1,b0})
           G<= 1;
      else
           G<= 0;
      // equivalent to G={a1,a0}>{b1,b0};

E<={a1,a0}=={b1,b0};  //logical equality
   end

endmodule
```