Laboratory Exercise 3

Combinational Logic and Displays

This is an exercise in designing combinational circuits that can drive 7-segment displays, and perform a variety of different functions.

Preparation Before the Lab

You are required to complete Parts I to III of the lab by writing and testing Verilog code and compiling it with Quartus II. Show your schematics, Verilog, and simulations for Parts I to III to the teaching assistants. You must simulate your circuit with ModelSim (using reasonable test vectors using the format shown in the previous lab).

In-lab Work

You are required to implement and test all of Parts I to III of the lab. You need to demonstrate all parts to the teaching assistants.

Part I

For this part of the lab, you will be learning how to use *always* blocks and *case* statements to design a 7 to 1 multiplexer. A module can contain any number of *always* blocks just the same as any module can contain any number of other module instantiations. The difference is that an *always* block can only instantiate logic within the module where it is defined. A module can be instantiated in any other module, i.e., it can be reused.

Fill out the rest of the case statement and add any wires you may need and connecting them using the **assign** statement described in the previous lab.

```
always @(*)                 // declare always block
begin
   case (select[2:0])       // start case statement
   begin
      0: out[0] = in[0];    // case 0
      1: ...                // case 1
      2: ...                // case 2
      3: ...                // case 3
      4: ...                // case 4
      5: ...                // case 5
      6: ...                // case 6
      default: ...          // default case
   end
end
```

An always block is triggered whenever there is a change in the sensitivity list. This list is denoted by the asterisk character in the above example. This means that whenever any input is changed, the following code will be simulated. Similarly, we can change the asterisk to certain inputs to limit when this code is triggered, but this can lead to simulations that do not match the real hardware. One of the (bad) features of the language. The accepted practice today is to always use the asterisk.

It is important to have a default case to ensure that all cases are covered. Otherwise, you can again have simulations that do not match the hardware. Yet another Verilog feature! Your goal is to write Verilog that will

generate hardware that exactly matches the simulation, so please put in the *default* statement.

If you want to know why, read on. When you execute an always block, the use of *if* and *case* statements can take you through different code paths. If you reach the end of the always block and there is an unassigned (reg) variable, then a memory element, a latch, will be created because the meaning is that the variable keeps its previous value, so a memory element is inferred. The problem becomes more subtle because if *select* in the above example is three bits, there are actually more than eight cases! Each bit can be (1,0,X,Z), so there are really 64 possible paths. Synthesis tools will likely assume only (1,0) and create the correct circuit, but the simulator may not do the same. Always, always put in the *default* statement.

Using SW_{6-0} as the data inputs SW_{9-7} as the select signals. Display on $LEDR_0$ the output of a 7 to 1 multiplexer using the case statement shown above.

1. Draw a schematic outlining the hierarchies you will use and explain them to the TA as part of your prelab.
2. Create a new Quartus II project for your circuit.
3. Include your Verilog file for the circuit in your project. Use switch SW_{9-7} on the DE1-SoC board as the $select$ input, switches SW_{6-0} as the data inputs. Connect the output to $LEDR_0$.
4. Simulate your circuit with ModelSim for different values of $select$ and $data$. You must show these to the TA as part of your prelab.
5. Compile the project.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Part II

Figure 2a shows a circuit for a *full adder*, which has the inputs a, b, and c_i, and produces the outputs s and c_o. Parts b and c of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum $c_o s = a + b + c_i$. Figure 2d shows how four instances of this full adder module can be used to design a circuit that adds two four-bit numbers. This type of circuit is usually called a *ripple-carry* adder, because of the way that the carry signals are passed from one full adder to the next. Write Verilog code that implements this circuit, as described below. Be sure to use what you learned about hierarchy in Lab 2.

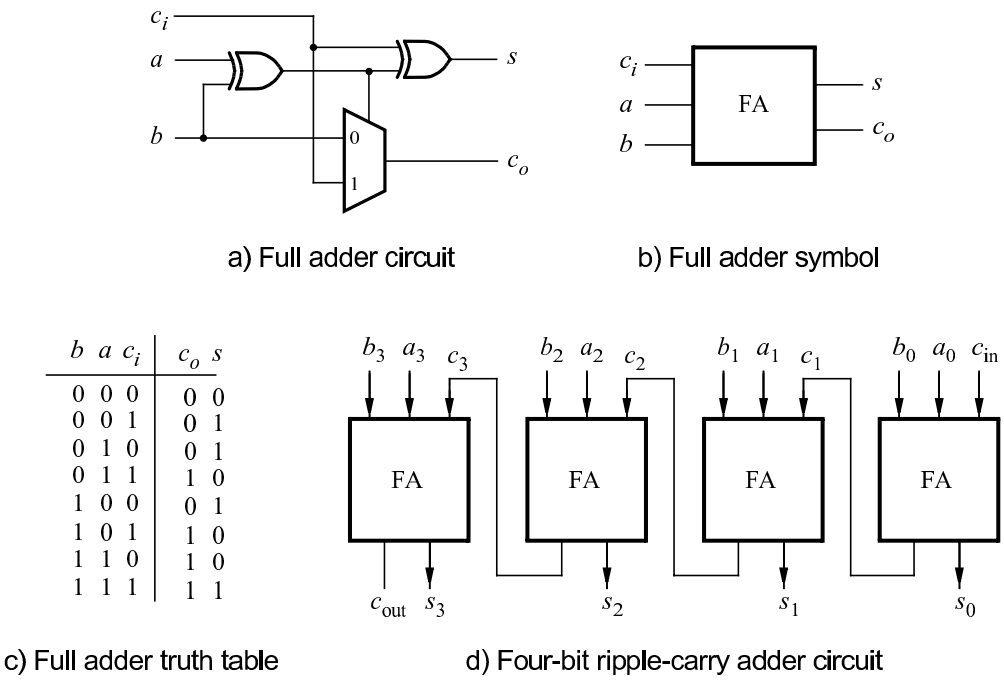

Figure 2. A ripple-carry adder circuit.

Perform the following steps.

1. Draw a schematic outlining the hierarchies you will use and explain them to the TA as part of your prelab.
2. Create a new Quartus II project for the adder circuit. Write a Verilog module for the full adder subcircuit and write a top-level Verilog module that instantiates four instances of this full adder.
3. Use switches SW_{7-4} and SW_{3-0} to represent the inputs A and B, respectively. Use SW_8 for the carry-in c_{in} of the adder. Connect the outputs of the adder, c_{out} and S, to the green lights $LEDR_9$ and $LEDR_{3:0}$ respectively.
4. Simulate your adder with ModelSim for intelligently chosen values of A and B and c_{in}. You must show these to the TA as part of your prelab. Note that as circuits get more complicated, you will not be able to simulate or test all possible cases. This means that you have to test only a subset. Here *intelligently chosen* means to find particular *corner cases* that exercise all aspects of the circuit. Be prepared to explain why your test cases are good enough.
5. Compile the project.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Part III

Using Parts I and II from this lab and the HEX decoder from Lab 2 Part III, you will implement a simple Arithmetic Logic Unit (ALU). This unit can perform multiple operations such as addition, sign extension, etc. The output result of each operation is sent to the multiplexer. If selected by the multiplexer, the operation result will display at the output.

Shown in the case statement blew are the operations for each case. The multiplexer has an 8-bit input and 8-bit output.

```
always @(*)          // declare always block
begin
   case (select)     // start case statement
   begin
      0: Addition using Part II of this Lab
      1: Addition using '+' operator
      2: Sign extend value B (SW3−0) to 8 bits
      3: Find if at least 1 of the 8 bits is 1 using a single OR operation
      4: Find it all of the 8 bits are 1 using a single AND operation
      5: Display the values on the switches
      default: ...   // default case
   end
end
```

Note that in this part of the lab, you will need to learn concatenation for the additions, sign extension, and reduction operations for ORing and ANDing multiple bits without typing ever single bit out.

Perform the following steps:

1. Draw a schematic outlining the hierarchies you will use and explain them to the TA as part of your prelab.
2. Create a new Quartus II project for your circuit.
3. Create a Verilog module for the simple ALU. Connect the A and B inputs to switches SW_{7-4} and SW_{3-0} respectively, and connect KEY_{2-0} for select signals. Display the outputs on $LEDR_{7-0}$; have *HEX0* and *HEX2* display output of A and B respectively and set *HEX1* and *HEX3* to 0. *HEX4* and *HEX5* should display the sum and carry out respectively.
4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. You must show this to the TA as part of your prelab.
5. Compile the project.
6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit.

Note: In your simulation, KEY_{3-0} are inverted. Remember that the DE1-SoC board recognizes an unpressed pushbutton as a value of 1 and a pressed pushbutton as a 0.