CSC258 - Lab 2

Multiplexers, Design Hierarchy, and HEX Displays

Fall 2016

Learning Objectives

The purpose of this exercise is to learn the importance of simulations and hierarchies when writing Verilog. We will use switches SW_{9-0} on the DE1-SoC board as inputs to the circuit. We will use light emitting diodes (LEDs) and 7-segment displays as output devices.

Note that we may refer to signals as SW_{9-0}, i.e., with the subscripts, but when you write your Verilog, you will need to use `SW[0]`, `SW[1]`, etc.

Resources

You can find many resources about the DE1-SoC board here `http://cd-de1-soc.terasic.com/`. The User Manual for the DE1-SoC board can be downloaded from here: `http://www-ug.eecg.toronto.edu/desl/manuals/DE1-SoC_User_manual.pdf`

Marking Scheme

Each lab is worth 4% of your final grade, but you will be graded out of 8 marks for this lab, as follows.

- Prelab + Simulations: 3 marks
- Part I (in-lab): 1 mark
- Part II (in-lab): 2 marks
- Part III (in-lab): 2 marks

Preparation Before the Lab

For this lab, and all future labs, you will be asked to prepare schematics (not in Quartus), Verilog code and ModelSim simulations for your prelab. The schematics should show the structure of your Verilog code, much like the schematics in Lab 1 showed how your circuit should be built. Your Verilog code will consist of a number of **modules** and the schematic should show how those modules are wired together, as well as the input and output ports of your circuit, i.e., connections to switches, LEDs, 7-segment hex displays, etc. Think of **modules** as *logic functions* consisting of multiple gates, such as the logic functions you wired together in Lab 1. All port names of the modules, wires and I/O ports should be clearly labeled in your schematics. Figure 2 below is an example.

Your Verilog code should be well-commented. For your simulations, you should have a script, or number of scripts, that test important aspects of your design. Print out the waveforms from the simulator and paste them into your lab book. If the simulation is very long, just print out enough to show that key parts of your circuit are working and as evidence that you have done the simulations. It is not necessary to have pages and pages of waveforms. However, occasionally, you will be asked to demonstrate and explain your entire simulation to the TA in the lab, so be prepared for this. As an example, if your circuit implements a logic function with three or four inputs, it is reasonable to show waveforms demonstrating the functionality of all possible combinations of input values. However, if your circuit implements a logic function with ten inputs, it would be unreasonable to simulate all 2^{10} possible input values.

Part I

Verilog File (.v):

The DE1-SoC board provides 10 toggle switches, called SW_{9-0}, that can be used as inputs to a circuit, and 10 red lights, called $LEDR_{9-0}$, that can be used to display output values.

A Verilog file for a 2-to-1 multiplexer, named *mux.v*, has already been provided to you. The top module *mux* has 3 inputs. SW[0] is the input 0 signal, SW[1] is the input 1 signal, and SW[9] is the select signal. The output is displayed on LEDR[0].

module mux (SW, LEDR); //module name and port list

The top module, *mux*, is a very trivial example of a design hierarchy, as it instantiates a single *mux2to1* module. In the more general case, any module can instantiate a number of interconnected modules, just like when you wired up a number of chips in Lab 1. However, in any circuit you build, there must be only one *top-level* module. The *.port(connection)* notation matches the port name from the *mux2to1* module to a connection (e.g., port or internal wire) inside the *mux* top-level module. Note that multiple modules of the same type can be used to build larger circuits. Each use of a module is called an *instance*. Every instance has to have a unique name. In our example below we used only one instance of the mux2to1 module, which we named *u0*.

```
mux2to1 u0 (
    .x(SW[0]),      // connect port SW[0] to port x
    .y(SW[1]),      // connect port SW[1] to port y
    .s(SW[9]),      // connect port SW[9] to port s
    .m(LEDR[0])     // connect port LEDR[0] to port m
);
```

Figure 1 shows the symbol for a 2-to-1 multiplexer. As mentioned in Lab 1, a multiplexer is a device that uses a select signal to select which one of multiple inputs should appear on the output of the device. In our example below, input *s* will control which of the inputs *x* and *y* will appear on the output *m*. If s is 0, *x* will appear on the output, while if s is 1, *y* will appear on the output.

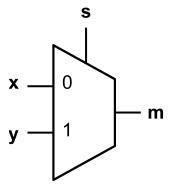

Figure 1: Symbol for a 2-to-1 multiplexer

The Boolean expression for a 2-to-1 multiplexer is `m = xs'+ ys`, and one way you can express this in Verilog is the following:

```
assign m = x & ~s | y & s;
```

Table 1 shows a mapping of all the bitwise Verilog operators to Boolean symbols.

Table 1: Verilog Operators

|	bitwise OR
&	bitwise AND
~	bitwise negation
^	bitwise XOR

Simulation File (.do):

After examining the file, to verify the code functions properly, we can perform a simulation using a script written in a *.do* file. This file is also provided by your instructor.

Inside the *.do* file, we start off by creating a working directory called *work* using the **vlib** command. We then compile the Verilog file using **vlog** and load it into the simulation with the **vsim** command. Lastly, to display all the signals on the waveform viewer, we put {/*} after **add wave**.

```
# Set the working dir, where all compiled Verilog goes.
vlib work

# Compile all Verilog modules in mux.v to working dir;
# could also have multiple Verilog files.
# The timescale argument defines the default time unit
# (used when no unit is specified), while the second number
# defines precision (all times are rounded to this value).
vlog -timescale 1ns/1ns mux.v

# Load simulation using mux as the top level simulation module.
vsim mux

# Log all signals and add some signals to waveform window.
log {/*}
# add wave {/*} would add all items in top level simulation module.
add wave {/*}
```

Once everything is initiated, we set the input signals to be a 1 or a 0 with the **force** command. The **force** command specifies the state of the inputs in the simulation, which mimics setting the inputs using switches in the lab. The **run** command instructs the simulator to simulate the behavior of the circuit for a specified time duration. During this time, the inputs to the circuit are assumed to have the state specified by the last **force** command. If there are inputs whose values have not been specified using the **force** command, their values will be assumed unknown. Depending on the simulator settings, you may see these values appear displayed using special colour in waveforms, and their values labeled as either *z* or *x*, instead of 0 or 1. *z* and *x* are special symbols used to indicate unknown values.

```
# Set input values using the force command, signal names need to be in {} brackets.
force {SW[0]} 0 # force SW[0] to 0
force {SW[1]} 1 # force SW[1] to 1
force {SW[9]} 0 # force SW[9] to 0

# Run simulation for a few ns.
run 10ns # run for 10 ns
```

Perform the following steps:

1. When you have familiarized yourself with the *.do* file, open ModelSim, and in the ModelSim's Transcript window (near the bottom) use the **cd** command to change to the directory where you placed the *wave.do* and *mux.v* files. Next, type **do** *wave.do* (or the file name you named your *.do* file). Look at the simulation, that is the generated waveform, and verify that the provided test-cases work as expected. **(PRELAB)**.

2. Create a new Quartus Prime project for the Verilog code provided and test it on the board during your lab session. Do not forget that you will need the `DE1_SoC.qsf` file to define how the switches and LEDs connect to the pins.

3. Compare the output results with the simulations you performed.

4. Did you notice a significant time difference between testing with ModelSim and implementing your design on the board? The difference becomes greater as the complexity of the circuit increases. Comment on this difference and its impact on debugging.

Part II

Start with the code given in Part I and modify the design to make it a 4-to-1 multiplexer. You must use multiple instantiations of the *mux2to1* module given to you in Part I. This is known as hierarchical design and is a good practice especially for larger designs where the Verilog code you write can become more difficult to debug. Smaller submodules are generally easier to test thoroughly and debug.

To complete this section, you will need to use the **wire** declaration to create wires that can be used to connect the multiple blocks together.

wire Connection; // defines a wire called Connection

The wire created above is called *Connection* and it can be used to connect the output of a module to the input of a module, the same way you used a physical wire in Lab 1 to connect the output of one gate to the input of another gate. Figure 2 shows a schematic of two modules using the wire *Connection*. You may also use *Connection* as input to other logic expressions within the module in which this wire is defined.

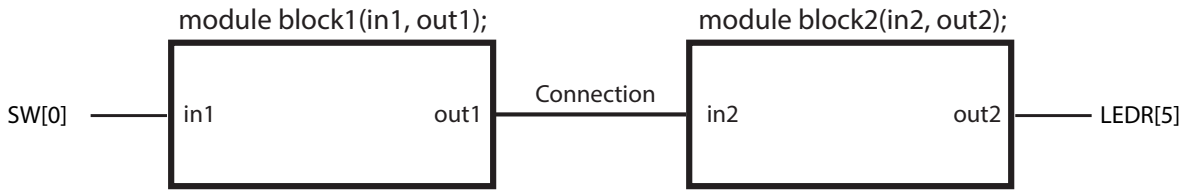

Figure 2: Using the wire *Connection* to make a connection between two modules

The following code fragment corresponds to Figure 2. It creates *instances* of modules *block1* and *block2*, named *B1* and *B2*, respectively. The wire *Connection* is used to wire the module instances together.

```
block1  B1 (
   .in1(SW[0]),          // connect external port SW[0] to port in1 of block1
   .out1(Connection)     // connect wire Connection to port out1 of block1
);

block2  B2 (
   .in2(Connection),     // connect wire Connection to port in2 of block2
   .out2(LEDR[5])        // connect external port LEDR[5] to port out2 of block2
);
```

Another way to make a connection is to use the assign statement. For example, if we wanted to connect the **wire** called *Connection* to $LEDR_0$, we do the following:

assign LEDR[0] = Connection; // joins wire Connection to LEDR[0]

Now construct a module for the 4-to-1 multiplexer shown in Figure 3 with the truth table shown in Table 2 using the **wire** construct and multiple instances of the mux2to1 module. Note that the truth table in Table 2 is given in a short-hand form. A real truth table would consist of rows enumerating all possible combinations of values of

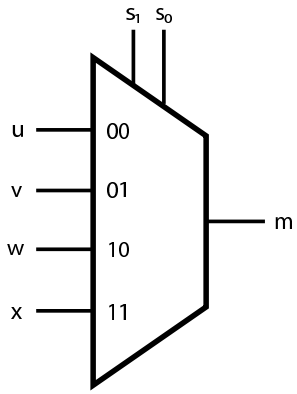

Figure 3: Symbol for a 4-to-1 multiplexer

Table 2: Truth table for a 4-to-1 multiplexer

$s_1 s_0$	m
00	u
01	v
10	w
11	x

inputs *u*, *v*, *w*, *x*, in addition to s_0 and s_1, and show the value (0 or 1) of the output *m* for each row of the truth table. Since this would result in a truth table with a large number of rows, it is written in short-hand form instead.

Perform the following steps:

1. Answer the following question: if the truth table in Table 2 was given in full, how many rows would it have? **(PRELAB)**
2. Draw a schematic (not in Quartus) showing how you will connect the *mux2to1* modules to build the 4-to-1 multiplexer. Be prepared to explain it to the TA as part of your prelab. The schematic should reflect how you are going to write your Verilog code. **(PRELAB)**
3. Create a new Quartus Prime project for your circuit and write the Verilog code. **(PRELAB)**
4. Include your Verilog file for the circuit in your project. Use switches SW_{9-8} on the DE1-SoC board as the 2-bit s input, and switches SW_{3-0} as the data inputs (labeled as *u*, *v*, *w*, *x* in Figure 3). Connect the output to $LEDR_0$. Do not forget that you will need the `DE1_SoC.qsf` file to define how the switches and LEDs connect to the pins.
5. Simulate your circuit with ModelSim for different values of s, u, v, w and x. Do enough simulations to convince yourself that the circuit is working. You must show these to the TA as part of your prelab. **(PRELAB)**
6. Compile the project.
7. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Part III

In this part of the lab, you are to design a decoder for the 7-segment HEX display as shown in Figure 4. The output of the HEX display is determined by the value at the input of the decoder as shown in Table 3. We call this a HEX display because it can display all hexadecimal digits.

The 7-segment display uses a *common anode*. What does *common anode* mean in terms of lighting up a segment? You should be able to find the answer online. Section 3.6.2 in the DE1-SoC User manual also tells you what is needed to turn on a segment.

HINT: In order to solve this part you need to first identify which segment needs to be illuminated for every input and then write a Boolean function for each one of the seven segments of the HEX display so they are turned on when needed. You must use K-maps to optimize those Boolean expressions before you write the corresponding Verilog code.

Table 3: Desired behaviour of HEX decoder

$c_3c_2c_1c_0$	Character
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	b
1100	C
1101	d
1110	E
1111	F

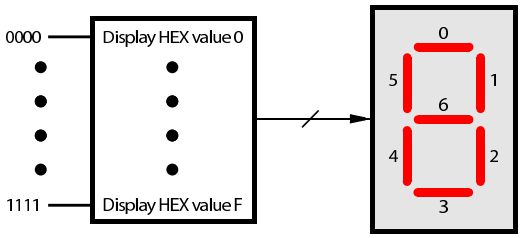

Figure 4: HEX decoder

Debugging Strategy: Before diving into the full Part III implementation, you may find it helpful to start by creating a very simple Verilog module which turns on segment 0 of the 7-segment display, i.e., pin $HEX0_0$, based on the input from SW_0.

Perform the following steps:

1. Write the expressions for seven Boolean functions, one for each segment of the 7-segment decoder. You have to use K-maps for optimization. You should be able to explain to the TA how you generated these expressions by showing them the truth-tables you wrote, and K-maps you used to optimize your circuits. **(PRELAB)**

2. Write a Verilog module for the 7-segment decoder taking advantage of the aforementioned expressions. **(PRELAB)**

3. Simulate your 7-segment decoder module with ModelSim, ensuring the output waveforms are correct. You should use the first four letters of each teammate's first name as an input to the simulation after you've substituted them with their corresponding keypad numbers[1]. For example, for first names **Myrt**o and **Fran**k, the initial numbers will be 6978 3726. Substitute any duplicate numbers with hexadecimal digits (A through F). Here the second instances of 7 and 6 should change to A and B respectively. Therefore, for this example, the input to the circuit in the simulation should be, over time, the binary numbers corresponding to the following hexadecimal digits: 6978 3A2B. You must show simulation waveforms to the TA as part of your prelab. **(PRELAB)**

4. Create a new Quartus Prime project for your circuit. You should instantiate the 7-segment decoder module in your top-level module. Connect the $c_3c_2c_1c_0$ inputs to switches SW_{3-0}, and connect the outputs of the decoder to the *HEX0* display on the DE1-SoC board. The segments in this display are called $HEX0_0$, $HEX0_1$, . . ., $HEX0_6$. You should declare a 7-bit port for the segments in your Verilog code, as follows:

   output [6:0] HEX0;

   This way, the names of these outputs match the corresponding names in the *DE1-SoC User Manual* and the pin assignment `DE1_SoC.qsf` file.

5. Compile the project.

6. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the SW_{3-0} switches and observing the 7-segment display.

[1]Use the letter to digit mapping from here: `https://en.wikipedia.org/wiki/File:Telephone-keypad2.svg`