Laboratory Exercise 2

Multiplexers, Hierarchy, and HEX Displays

September 18, 2017

The Altera University web site at https://www.altera.com/support/training/university/materials-tutorials.html has tutorials that you should read on your own. For this lab, you should do the Verilog version of *Quartus Introduction*. Perform the tutorial steps outside of the lab using simulation only.

The purpose of this exercise is to learn the importance of simulations and hierarchies when writing in 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.

Preparation Before the Lab

For this lab, and all future labs, you will be asked to prepare schematics, Verilog code and ModelSim simulations in your preparation. 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 the modules are wired together, and the input and output ports of your circuit, i.e., connections to switches, LEDs, displays, etc. Think of **modules** as just complex *gates*, such as the gates you wired together in Lab 1. All port names of the modules, wires and I/O ports should be clearly labeled. Figure 1 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.

For this lab do the following preparation:

Part I Do all of the simulation parts and prepare a bitstream that you can test when you go to the lab. You do not need to demonstrate your circuit working, but you may be asked questions about the preparation.

Part II and Part III You are required to write the Verilog code for both parts. For marking of the preparation, you are required to show the teaching assistants your schematics, Verilog code, and ModelSim simulations for Parts II and III.

In-lab Work

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

Part I

In this part you are provided with two files: a Verilog file with a design example and a simulation script to show some basic commands for simulating the design.

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 has been provided for a 2 to 1 multiplexer. 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

Note that the above module definition for *mux* actually declares more than 3 inputs and 1 output, contrary to the above statement. By using SW and LEDR in the module port definition, you actually declare all switches SW_{9-0} as inputs and all $LEDR_{9-0}$ as outputs of the module, even though only a subset of the inputs and outputs are used. It is just being lazy, and does not affect the circuit created. Strictly speaking, according to the original statement, the declaration should have been:

module mux (SW[0], SW[1], SW[9], LEDR[0]); //module name and port list

The top module, *mux*, is a very trivial example of using hierarchy where it instantiates a single *mux2to1* module uniquely identified as instance *u0*. 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)* statements match the port names defined in the *mux2to1* module to the connections inside the *mux* module. Think of the port name as the pin on a chip and you are connecting wires in the *mux* module to the pins of the chip, i.e., the ports of the *mux2to1* module instance.

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

Simulation File (.do):

After examining the Verilog file to understand what it is supposed to do, it is time to verify that the code functions properly. The Verilog file describes the structure and behaviour of a circuit. Before actually building the circuit, it is important to determine whether the Verilog description actually does what you intend. This is done by *simulation* of the circuit, which is done prior to actually building the circuit and testing it for real. 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
vlog 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 can set the input signals to be a 1 or a 0 with the **force** command and run the simulation for *x ns* with the **run** command.

```
# 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
```

When you have familiarized yourself with the *.do* file, open ModelSim, and in the terminal window (near the bottom) change to the file's working directory using the **cd** command and type **do** *wave.do* (or the file name you named your *.do* file).

Look at the simulation. You might be wondering how the time intervals are determined at this point. If we open the Verilog file again, we can see that the very first line states the timescale with the time unit and time precision. All time values are read as the time unit which is rounded to the nearest time precision.

Perform the following steps:

1. Run the default *.do* file given by your instructor.
2. Create your own test cases for the *.do* file and demonstrate that it works.
3. Create a new Quartus 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.
4. Compare the output results from the board with the simulations you performed.
5. Did you notice a significant compilation time difference between ModelSim and the actual on board test results? 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, 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 can become more difficult to 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; //creates 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 1 shows a schematic of two modules using the wire *Connection*.

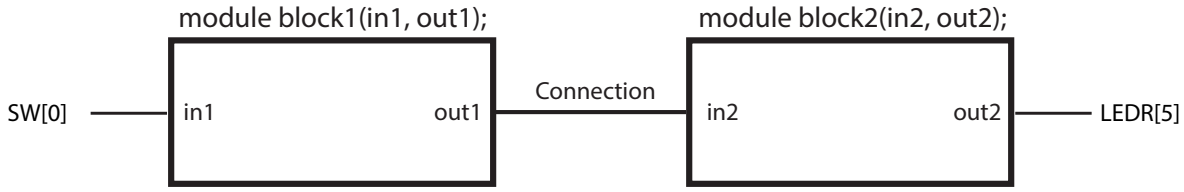

Figure 1: Using the wire *Connection* to make a connection between two modules

The following code fragment corresponds to Figure 1. It creates *instances* of modules *block1* and *block2*, named *B1* and *B2*, respectively. When using hierarchy, think of a module definition as the pattern for a sub-circuit and the definition of an *instance* of a module as creating an actual copy of that sub-circuit that you can use. You can create as many instances of a module as you need, which is like taking a number of the same type of chip from the cupboard and wiring them together to make a larger circuit. Here, *B1* is a sub-circuit that has the functionality defined by module *block1*. The wire *Connection* is used to wire the module instances together.

```
wire Connection;          //Declare the wire called Connection

...                       //Other stuff

block1  B1 (
   .in1(SW[0]),           // assign port SW[0] to port in1
   .out1(Connection)      // assign wire Connection to port out1
);

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

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 2 with the truth table shown in Table 1 using the **wire** construct and multiple instances of the mux2to1 module.

Table 1: Truth table for 4 to 1 multiplexer

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

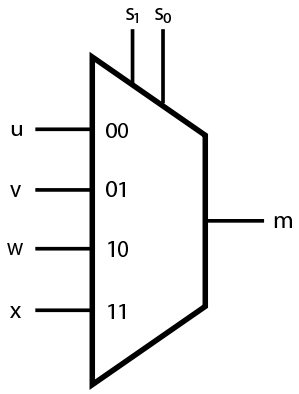

Figure 2: Symbol for 4 to 1 multiplexer

Perform the following steps.

1. Draw a schematic showing how you will connect the *mux2to1* modules to build the 4 to 1 multiplexer. How many *mux2to1* instances will you need? Your schematic should have a top-level module that represents the 4 to 1 multiplexer and inside the top-level module you should instantiate the number of 2 to 1 multiplexers you need. Give names to all the wires that you need to make the connections and also give names to all the instances of the 2 to 1 multiplexers. Be prepared to explain your schematic to the TA as part of your prelab. The schematic should reflect exactly how you are going to write your Verilog code.
2. After drawing your schematic, write the Verilog code that corresponds to your schematic. Your Verilog code should use the same names for the wires and instances shown in the schematic.
3. Create a new Quartus project for your circuit.
4. Include your Verilog file for the circuit in your project. Use switches SW_{9-8} on the DE1-SoC board as the s input, switches SW_{3-0} as the inputs. 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.
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.

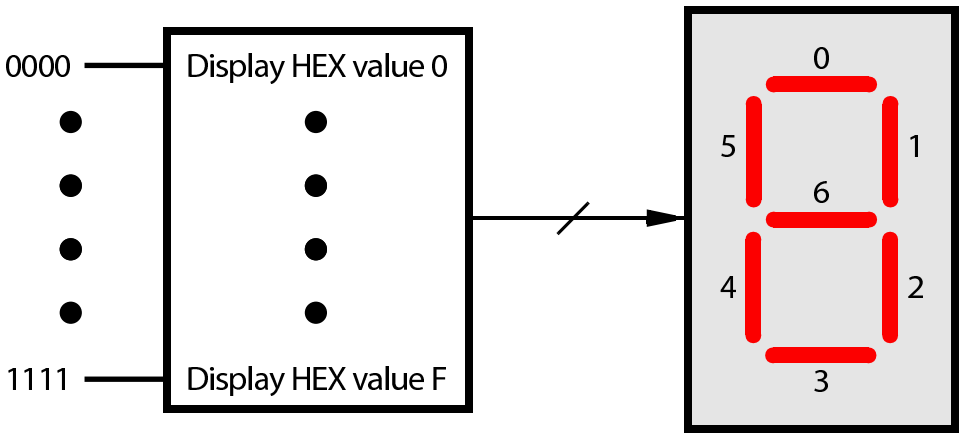

Figure 3: HEX decoder driving a HEX display

Part III

In this part of the lab, you are to design a HEX decoder for the 7-segment HEX display as shown in Figure 3. In general, a decoder is a circuit that takes an input pattern and converts (decodes) it into a different pattern. A HEX decoder determines how to drive a HEX display according to the value at the input of the decoder as shown in Table 2. You should create a module for the HEX decoder so that you can instantiate it each time you need to drive a HEX display, i.e., instantiate one HEX decoder for every HEX display you are using. You will also use the HEX decoder in future labs because we always want to display some kind of output number.

To figure out the circuit for the HEX decoder, work through the following steps:

1. How many inputs to the decoder are there and what are they connected to?
2. How many outputs from the decoder are there and what are they connected to? Note that each segment in the HEX display can be individually controlled with a separate input to the HEX display.
3. Knowing the inputs and outputs of the HEX decoder will tell you what input and output ports should be in your module declaration for the HEX decoder.
4. For each of the outputs of your decoder, derive the truth table and then the logic expression corresponding to the truth table. For example, consider Segment 1. For each row in the truth table for Segment 1 figure out whether Segment 1 should be turned on or off. Do this for every segment. You should end up with a logic expression for every segment. These are the equations that will be inside your HEX decoder module.

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.

Table 2: Truth table for 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

Perform the following steps:

1. Draw a schematic of the circuit for the HEX decoder module and be prepared to explain it to the TA as part of your prelab. You do not need to draw gates for each of your equations. In Verilog, you are just going to input the equations, so your schematic can just refer to *Eqn 0, Eqn 1, ...* where the equations are defined from your derivations. Note that you do not have to simplify your equations. The tool will do that for you!

   The schematic should reflect how you are going to write your Verilog code, so be sure to use the same names for your signals in the schematic and yor Verilog code. It can make debugging easier.

2. Create a new Quartus project for your circuit.

3. Create a Verilog module for the 7-segment decoder. Instantiate your decoder inside a top-level module. The top-level module should have the $c_3c_2c_1c_0$ inputs connected to switches SW_{3-0}, and the outputs of the decoder connected 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 the 7-bit port like this:

   output [6:0] HEX0;

   in your Verilog code so that the names of these outputs match the corresponding names in the *DE1-SoC User Manual* and the pin assignment `DE1_SoC.qsf` file.

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 by toggling the SW_{3-0} switches and observing the 7-segment display.