Laboratory Exercise 2

Multiplexers, Hierarchy, and HEX Displays

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.

Preparation Before the Lab

You are required to write the Verilog code for Parts II and III of the lab. For marking of preparation by the teaching assistants, 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 to III of the lab. You need to demonstrate both parts to the teaching assistants.

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 has been provided by your instructor 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

The top module, *mux*, is a very trivial example of using hierarchy where 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)* matches the port from the *mux2to1* module to the connection inside the *mux* module.

```
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 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
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 familarized 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 as part of your prelab.

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 II project for the Verilog code provided and test it on the board during your lab session.
4. Compare the output results with the simulations you performed.
5. Did you notice a significant compilation time difference between ModelSim and the actual on board test results? Also, comment why.

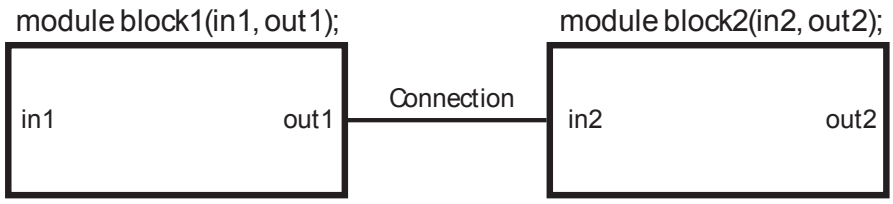
Part II

Using the code given in part 1, scale the design such that it is a 4 to 1 multiplexer. You must use multiple instantiations of the *mux2to1* given to you in part 1. This is known as hierarchical designs and is a good practice especially for larger designs where the Verilog can become more difficult to debug.

In order to complete this section, you will need to create a **wire** to connect the multiple blocks together.

wire Connection; //creates a wire called ~~connection~~

The wire created above is called *Connection* and in order to connect two different blocks together such as shown below we can replace one of the input and output shown in part one:



The following code corresponds to the figure above:

```
block1  B1 (
    .in1(in);               // assign port in to port in1
    .out1(Connection);      // assign wire connection to port out1
);

block2  B2 (
    .in2(Connection);       // assign wire connection to port in2
    .out2(out);             // assign port out 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 following 4 to 1 multiplexer using the wire construct and multiple instances of the mux2to1 module.

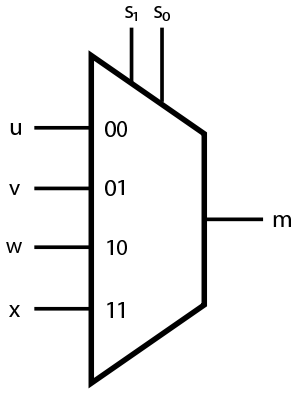


Figure 1. Multiplexer 3 to 1 example. (NOTE: you are doing a 4 to 1 Multiplexer)

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

Table 1. Multiplexer 4 to 1 output.

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. Include your Verilog file for the circuit in your project. Use switch SW_{9-8} on the DE1-SoC board as the s input, switches SW_{3-0} as the inputs. Connect the output to $LEDR_0$.
4. Simulate your circuit with ModelSim for different values of s, X, and Y. 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 III

In this part of the lab, you are to design a decoder for the 7-segment HEX display.

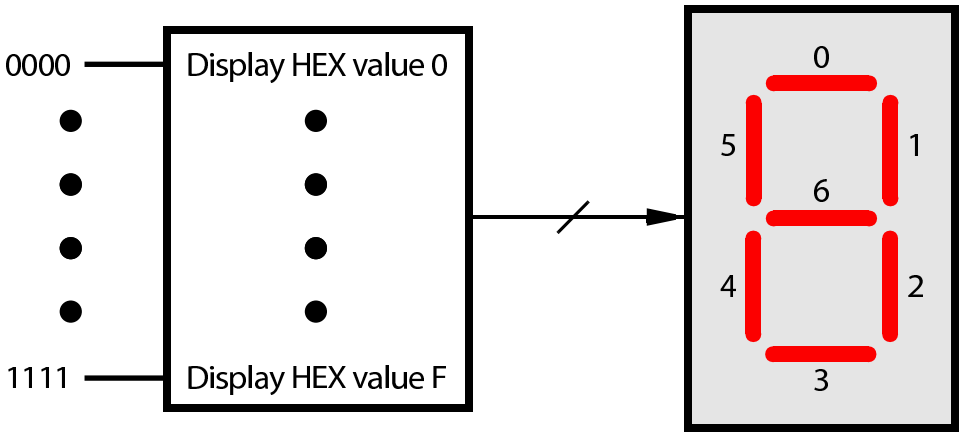


Figure 2. HEX Decoder.

$c_3c_2c_1c_0$	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

Table 2. HEX character codes.

Perform the following steps:

1. Draw a schematic and explain it 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 7-segment decoder. 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 the 7-bit port

   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.