

Laboratory Exercise 2

Multiplexers, Hierarchy, and HEX Displays

Revision of September 19, 2023

In this lab you will start to use Intel Quartus Prime to implement Verilog designs on FPGAs. If you have not done this already, go to the Intel University web site at <https://www.intel.com/content/www/us/en/developer/topic-technology/fpga-academic/materials.html> and download the Verilog version of *Introduction to Intel® Quartus® Prime Software (standard or lite)* under the *Tutorials* link. Without the board, you can still do everything up to programming, configuring and testing the design on the FPGA.

In this lab you will also learn some good practices for hardware design. In particular, you will learn how to do basic simulations and how to use hierarchies when writing in Verilog. The HEX decoder for the 7-segment display that you build in Part III will be used in most future labs.

1 Work Flow

For this lab, and all future Verilog labs, you should prepare schematics, Verilog code and ModelSim simulations. You should be prepared to present these to your TA if you are asking for help as they are the best way to show what you are trying to do and help the TA understand your work.

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., the connections where your circuit connects to other modules or signals. 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 a 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. Occasionally, you may be asked to demonstrate and explain your entire simulation when consulting a TA, so be prepared for this.

2 The Importance of Simulation

When doing hardware design for FPGAs and ASICs, simulation is extremely important. When building ASICs, where fabrication costs can be millions of dollars to build a chip, you clearly want to get it right the first time! Simulation is the only way to give you a chance. In this course, you will be using FPGAs, which are reprogrammable, so there will be the temptation to just load a design and try it, just as you do with your C programs. However, even a small design can take minutes to generate a programming bitstream, and large designs can take days. You will soon experience this slowness. It is a complaint that has existed since FPGAs were invented. Imagine debugging your C program if you could only test it once a week! Debugging with simulation is much closer to debugging your C program and you have much better visibility to see what is going on in your design. Once your design is running in an FPGA, you have few options to see what is going on in your circuit.

In these labs, there will be a strong emphasis on doing simulation first. You should not even try running your design in real hardware unless your simulations work first. Remember this golden rule: *If you don't simulate, it won't work. If you do simulate, it might work.* Note, there are no guarantees in this business!

3 Automarker

This lab will be graded with an Automarker on the UG machines. Instructions for using the Automarker are provided in the *Automarker Submission Instructions*. Please take care to follow the module signatures (declarations) given in the lab so that your submission can be properly graded. Also, there are two versions of the Automarker: one for ECE241 and one for ECE253. **Be sure to submit your work to the ECE241 automarker. Incorrectly submitted work will not be graded!** You can, before the deadline, re-submit as many times as you need. The submission command is:

```
/cad2/ece241f/public/submit 2 part2.v part3.v
```

Assignments submitted with an incorrect command **will not be graded** (e.g., wrong course or lab number). You can check if it was submitted using the command:

```
/cad2/ece241f/public/check_submission 2
```

You may also run a sanity check using the command

```
/cad2/ece241f/public/2/tester
```

Note the tester only tests a couple trivial cases to make sure the file and module names are correct, it is not a full test of your code. You need to run simulations.

4 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.

4.1 Verilog File

Examine the `mux.v` file provided to you. You will see two modules `mux` and `mux2to1`. The fundamental construct for defining a block of circuitry is the *module* (textbook Section A.8). The `mux2to1` module defines the main functionality of this circuit as it describes a 2-to-1 multiplexer component (textbook Section 2.8.2).

As a component, the `mux2to1` module can be *instantiated* (copied) as many times as needed, just as you might require multiple 7408 chips when you need many AND gates. The `mux` module is what we call a *top-level* module, which describes the overall circuit being built. The *top-level* module includes all the *pins* of your circuit, like the pins of the 7404 chip, i.e., how to get signals into and out of the overall circuit.

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

We will use the switches and LEDs as a way to generate inputs to your circuit and observe outputs from your circuit. In this example, the `mux` module describes connections from a 2-to-1 multiplexer *instance*, called `u0`, to the switches and LEDs of the DE1-SoC board. The connections used are shown in Table 1

u0 Port Name	Direction	DE1-SoC Pin Name
x	Input	SW[0]
y	Input	SW[1]
s	Input	SW[9]
m	Output	LEDR[0]

Table 1: u0 port mapping to DE1-SoC pin names

The top-level module, `mux`, is defined with the following signature:

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

Note that the module definition for *mux* actually declares more than 3 inputs and 1 output, contrary to the actual requirements. 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 trivial example of using hierarchy (textbook Section 2.10.3) 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. When instantiating, you can use *.port(connection)* statements that 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]),    // connect port SW[0] to port x of mux2to1
    .y(SW[1]),    // connect port SW[1] to port y of mux2to1
    .s(SW[9]),    // connect port SW[9] to port s of mux2to1
    .m(LEDR[0])   // connect port LEDR[0] to port m of mux2to1
);
```

4.2 Simulation File (wave.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 to you.

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 {} parentheses
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 that is rounded to the nearest time precision.

4.3 Building the Design for FPGA

Now that you have tried simulating a Verilog file, you will now build the mux for programming the FPGA.

1. Run the provided *.do* file.
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 just using ModelSim and generating the bitstream for programming the FPGA? The difference becomes greater as the complexity of the circuit increases. Comment on this difference and its impact on debugging.

4.4 Part II

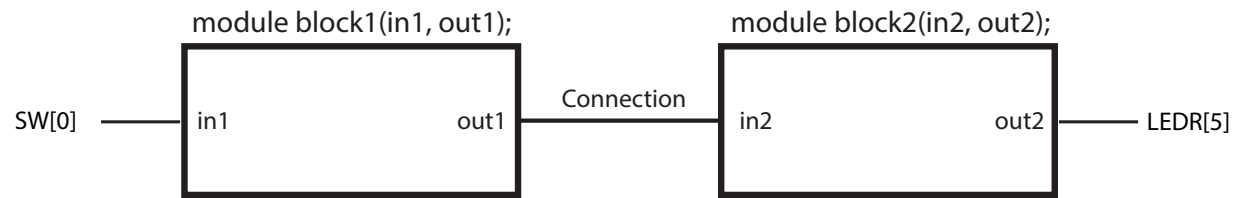


Figure 1: Using the wire *Connection* to make a connection between two modules

In this part, you will write Verilog to implement modules with the functionality of three 7400-series chips from Lab 1. Then, you will use the Verilog modules of these chips to implement the 2-to-1 multiplexer in Part I above. Part of the rationale for this exercise is to emphasize that the hardware functionality within the discrete 7400-series chips you used in Lab 1 can also be realized on an FPGA. In essence, the FPGA becomes the “breadboard” where you build circuits and interconnect them with one another.

4.4.1 The Wire

To implement your design, you will need to use the **wire** declaration (textbook Section A.6.1) 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

```
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
);
```

Figure 2: Code fragment for the circuit in Figure 1.

the output of one gate to the input of another gate. Figure 1 shows a schematic of two modules using the wire *Connection*.

The Verilog code fragment shown in Figure 2 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 chips 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*. *Connection* is used to wire the module instances together.

Another way to make a connection is to use the `assign` statement. For example, if we wanted to connect the **wire** called *Connection* to *LEDR₀*, we do the following:

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

4.4.2 How to build the 2-to-1 multiplexer

Write three separate Verilog modules for the following three 7400-series chips from Lab 1:

- 74LS04/05 (hex inverters)
- 74LS08/09 (quad 2-input AND gates)
- 74LS32 (quad 2-input OR gates)

Refer to the Lab 1 handout for the pin outs and functionality of the above chips.

Here is an example module declaration for the 74LS04/05:

```
module v7404 (pin1, pin3, pin5, pin9, pin11, pin13,  
             pin2, pin4, pin6, pin8, pin10, pin12);
```

Use the same pin ordering for the `v7408` and `v7432` modules. For a hint, see Chapter 3 in the *Handbook of Parts for ECE241/253 Circuits*, which has a complete module definition example for a 74LS00/03 (four 2-input NAND gates). Note that you do not need to add power/ground pins to the module declarations, as you will be implementing their functionality on an FPGA chip.

After writing and simulating the three 7400-series modules, write another Verilog module that implements a 2-to-1 multiplexer using the three 7400-series modules. The top-level module declaration should be

```
module mux2to1 (x,y,s,m);
```

with the same definitions for the ports as used in Part I. You will need one instance of each of your three 7400-series modules. The reason is that the logic function for the 2-to-1 multiplexer is: $m = \bar{s}x + sy$, which requires NOT, AND, and OR. Note that although the 2-to-1 multiplexer will not use *all* of the internal gates of each 7400-series module, the 7400-series modules should be complete and contain all of the functionality of the original corresponding 7400-series chip. Using “named-port” instantiation (the style used to create the `u0` instance of `mux2to1` in Section 4.1) will allow you to ignore the unused gates in the 7400-series chips in your top-level design of the 2-to-1 multiplexer.

Perform the following steps.

1. First draw a schematic. The top-level module in your schematic will represent the complete 2-to-1 multiplexer circuit. Inside the top-level module is where you instantiate each of the 7400-series modules and connect them to build the 2-to-1 multiplexer. The connections of the three modules should look much like your schematic from Lab 1.

Give names to all the wires that you need to make the connections and also give names to all the instances of the 7400-series modules. Show the names of each pin that you use on each module. The schematic will reflect exactly how you are going to write your Verilog code. For an example, look at Schematic 1 in Section 4.3.2 of the Handbook.

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. In total, you should have *four* Verilog modules with the signatures shown in Section 5.1 so that they can be graded with the Automarker.
3. Simulate the 7400-series modules and the top-level module with ModelSim for different input values. Do enough simulations to convince yourself that the circuit is working. You should have *four* simulations: one for each of the 7400-series modules, and one for the top-level module. Meaning, you should have four `.do` files, each simulating a different module (use the `vsim` command in the `.do` file to define which module to simulate). When you are satisfied with your simulations, you can submit to the Automarker.
4. Create a new Quartus project for your circuit.
5. Use `mux.v` from Part I but replace the `mux2to1` module in Part I with your new `mux2to1` written in this part. Do not forget that you will need the `DE1.Soc.qsf` file to define how the switches and LEDs connect to the pins.

6. Compile the project with Quartus to generate your bitstream. Make sure that there are no errors. If there are no errors your code is synthesizable, meaning it can create a hardware circuit and be downloaded into a board.
7. You can test your circuit by downloading the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

4.5 Part III

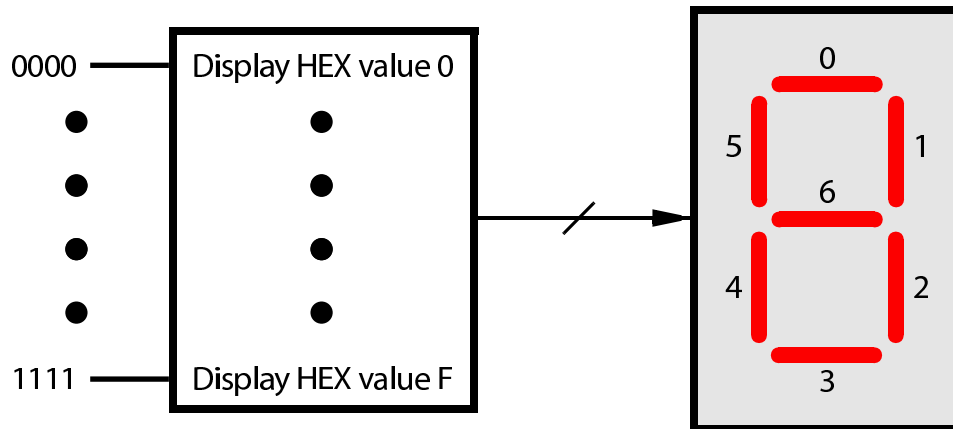


Figure 3: HEX decoder driving a HEX display

In this part of the lab, you are to design a HEX decoder for the 7-segment HEX display (textbook Section 2.8.3) 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.

Note: For this lab and all future labs, please form the numbers on the 7-segment display in the format shown in Figure 4. The letters should be formed according to 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 design for the HEX decoder, work through the following steps first:

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.



Figure 4: 7-Segment Format

Table 2: Truth table for HEX decoder

c_3	c_2	c_1	c_0	Character
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	b
1	1	0	0	C
1	1	0	1	d
1	1	1	0	E
1	1	1	1	F

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 on the DE1-SoC board 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.

To build your design, perform the following steps:

1. Draw a schematic of the circuit for the HEX decoder module. 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 your Verilog code. It will make debugging easier.

2. Create a Verilog module for the 7-segment decoder. Instantiate your decoder inside a module with the following signature:

```
module hex_decoder(c, display);
```

3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. When you are satisfied, you can submit your module to the Automarker.
4. Create a new Quartus project for your circuit. You will need a top-level module to make connections from the instantiation of the `hex_decoder` module to the switches and displays of the DE1-SoC board. The required mapping is shown in Table 3.

module Port Name	Direction	DE1-SoC Pin Name
<code>c</code>	Input	SW[3:0]
<code>display</code>	Output	HEX0[6:0]

Table 3: Module port mapping to DE1-SoC pin names

The individual segments of the 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.

5. Compile the project to generate a bitstream and make sure there are no errors.
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.

5 Submission

When submitting to the Automarker make sure you have modules declared as shown below as the Automarker will be looking for modules with these exact signatures.

5.1 Part II

For Part II, you need to submit a file named `part2.v` with the following modules in it:

1. `module v7404 (pin1, pin3, pin5, pin9, pin11, pin13, pin2, pin4, pin6, pin8, pin10, pin12);`
2. `module v7408 (pin1, pin3, pin5, pin9, pin11, pin13, pin2, pin4, pin6, pin8, pin10, pin12);`
3. `module v7432 (pin1, pin3, pin5, pin9, pin11, pin13, pin2, pin4, pin6, pin8, pin10, pin12);`
4. `module mux2to1(x, y, s, m);`

5.2 Part III

For Part III, you need to submit a file called `part3.v` with the following module in it:

1. `module hex_decoder(c, display);`