Laboratory Exercise 2

Hierarchy, Multiplexers, and Case Statements

Revision of September 15, 2025

In the previous lab, you built circuits using the gates present inside the 7400-series chips. In this lab, you will learn how to describe circuits using the System Verilog Hardware Description Language (HDL). You will also learn some good practices for hardware design, including how to do basic simulations and how to use hierarchies when writing in System Verilog. The simulation tool you will use to test your code's functionality before submission is ModelSim. Finally, you will have the option to run your code on the FPGA boards.

1 Work Flow

For this lab, and all future System Verilog labs, you should prepare schematics, System 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 System Verilog code, much like the schematics in Lab 1 showed how your circuit should be built.

Your System 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 System Verilog code should be well-commented.

2 The Importance of Simulation

When doing hardware design for FPGAs and ASICs,[1] 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. This course will place a strong emphasis on using simulations to verify the functionality of your designs.

[1] Application Specific Integrated Circuits

Once you confirm your simulations are working as expected, you will have the option to run your design on the FPGA boards. If you choose to do so, you will notice that even small designs, like those in this course, may take minutes to generate a programming bitstream, while larger designs can take days.[2] This is another reason why we emphasize using simulations. 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.

3 Automarker

This lab will be graded with an Automarker. 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.

4 Pre-lab

Before coming to lab, make sure to familiarize yourself with the videos provided in the Tutorial Videos for Lab 2 page. These videos contain important information regarding how to run the ModelSim simulator as well as how to compile and load your design onto the FPGA board.

Draw any schematics needed for the circuits in Part 2 and Part 3.

You should be prepared to show/explain your schematics (Part II step 1 and Part III step 1) to your TA at the beginning of the lab, and ready to answer questions from your TA. Both the pre-lab preparation and the answering of TA questions will contribute to your in-lab marks. Your online submission will contribute the rest of your lab mark.

5 Part I

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

[2] The programming bitstream is how your design is downloaded to FPGA board.

5.1 System Verilog File

You are provided with a `mux.sv` on Quercus, which can also be seen in Listing 1. In the file, you will see two modules `mux` and `mux2to1`. The fundamental construct for defining a block of circuitry is the *module* (textbook Chapter 4.1.1). The `mux2to1` module defines the main functionality of this circuit as it describes a 2-to-1 multiplexer component (textbook Chapter 4.2.4).

```
`timescale 1ns / 1ns // `timescale time_unit/time_precision

// SW[2:0] data inputs
// SW[9] select signals

// LEDR[0] output display

module mux(input logic [9:0] SW, output logic[9:0] LEDR);
    mux2to1 u0(
        .x(SW[0]),
        .y(SW[1]),
        .s(SW[9]),
        .m(LEDR[0])
        );
endmodule

module mux2to1(input logic x, input logic y, input logic s,
   output logic m);
    // x: select 0
    // y: select 1
    // s: select signal
    // m: output

    // assign m = s & y | ~s & x;
    // OR
    assign m = s ? y : x;

endmodule
```

Listing 1: 4:1 mux.sv. Note: do not copy paste text from the lab handout as unexpected characters can appear, leading to unexpected errors. Instead download the provided file from Quercus.

As a component, the `mux2to1` module can be *instantiated* 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 top-level module, `mux`, is defined with the following signature:

module mux (input logic [9:0] SW, output logic [9:0] LEDR); //module name and port list

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. Switches 0 (SW[0]) and 1 (SW[1]) are used as the x and y connections to the `mux2to1` instance `u0`, while switch 9 (SW[9]) is used as the select signal s, and LEDR[0] is used as the output signal.

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 (input logic SW[0], SW[1], SW[9], output logic LEDR[0]);

//module name and port list

The top module, `mux`, is a trivial example of using hierarchy where only one module instance `u0` is instantiated. In the more general case, any module can instantiate multiple 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
);
```

5.2 Simulation File (wave.do)

After examining the System Verilog file to understand what it is supposed to do, it is time to verify that the code functions properly. The System Verilog file describes the structure and behaviour of a circuit. Before actually building the circuit, it is important to determine whether the System Verilog description actually does what you intend. This is done by *simulation* of the circuit. 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 System 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 System Verilog goes
vlib work


# compile all System Verilog modules in mux.sv to working dir
# could also have multiple System Verilog files
vlog mux.sv


# 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

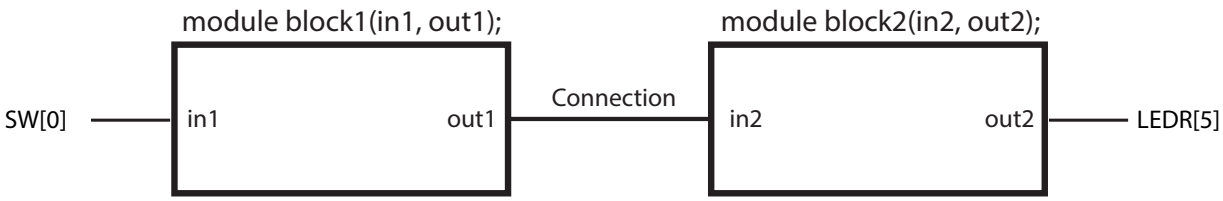

Figure 1: Using the wire *Connection* to make a connection between two modules

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 System 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.

5.3 Part II

In this part, you will write System Verilog to implement modules with the functionality of three 7400-series chips from Lab 1. Then, you will use the System 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 using a hardware description language such as System Verilog.

5.3.1 Connecting modules

To implement your design, you will need to use the **logic** declaration to create wires that can be used to connect the multiple blocks together. You can think of **logic** as a 'datatype' which indicates that you are creating a wire to connect things together.

logic 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*.

The System Verilog code fragment shown in Figure 2 corresponds to Figure 1.

```
logic 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.

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 $LEDR_0$, we do the following:

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

5.3.2 How to build the 2-to-1 multiplexer

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

- 74LS04/05 (six inverters)
- 74LS08/09 (four 2-input AND gates)
- 74LS32 (four 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 (input logic pin1, pin3, pin5, pin9, pin11, pin13,
              output logic pin2, pin4, pin6, pin8, pin10, pin12);
```

Use the same pin ordering for the `v7408` and `v7432` modules. Note that you do not need to add power/ground pins to the module declarations.

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

```
module mux2to1 (input logic x,y,s, output logic 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 5.1) 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 (to be completed before coming to lab). 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 System Verilog code.
2. After drawing your schematic, write the System Verilog code that corresponds to your schematic. Your System Verilog code should use the same names for the wires and instances shown in the schematic. In total, you should have *four* System Verilog modules with the signatures shown in Section 6.1 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.

5.4 Part III

In the previous section, you learned how to build a 2-to-1 multiplexer using either the `assign` statement or using modules that represent the 7400 series chips. It is important to note that System Verilog also provides a built-in method for designing multiplexers using the `case` statement. Listing 2 demonstrates how to implement a 4-to-1 multiplexer using a `case` statement.

In SystemVerilog, the `case` statement must be declared inside an `always_comb` block. The `always_comb` block works similarly to the `assign` statement used in the earlier `mux2to1` module. Therefore, you cannot use an `assign` statement inside the `always_comb` block because the block itself functions like an `assign` statement.

When using a `case` statement, it is essential to include a default case to ensure all scenarios are accounted for. For example, in a 4-to-1 multiplexer, while cases 0b00, 0b01, 0b10, and 0b11 are covered, undefined states like 0bxx or high impedance states like 0bzz could still occur.

```
module mux4to1(input logic [1:0] MuxSelect, input logic [3:0]
   MuxIn, output logic Out);

  always_comb // declare always_comb block
  begin
    case (MuxSelect) // start case statement
      2'b00: Out = MuxIn[0]; // Case 0
      2'b01: Out = MuxIn[1]; // Case 1
      2'b10: Out = MuxIn[2]; // Case 2
      2'b11: Out = MuxIn[3]; // Case 3
      default: Out = 0; //Default Case set arbitrarily
    endcase
  end
endmodule
```

Listing 2: 4:1 multiplexer

In this part of the lab, you will design a 7-to-1 multiplexer. Using the explanation given to you to create a 4-to-1 mux, create a 7-to-1 mux in the same fashion. To do so perform the following steps

1. Draw a schematic of the 7-to-1 multiplexer with all the wires, inputs and outputs clearly labeled. This should be completed prior to coming to lab.
2. After drawing your schematic, write the code that corresponds to your schematic. Your module declaration should be as follow:

```
mux7to1(input logic [2:0] MuxSelect, input logic [6:0] MuxIn, output
                                logic Out);
```

3. Simulate your circuit with ModelSim for different values of MuxSelect and MuxIn and submit your files to the Automarker when satisfied with your simulations.

6 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.

6.1 Submitting Part II

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

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

module v7408 (input logic pin1, output logic pin3, input logic pin5, input
logic pin9, output logic pin11, input logic pin13, input logic pin2, input
logic pin4, output logic pin6, output logic pin8, input logic pin10, input
logic pin12);

module v7432 (input logic pin1, output logic pin3, input logic pin5, input
logic pin9, output logic pin11, input logic pin13, input logic pin2, input
logic pin4, output logic pin6, output logic pin8, input logic pin10, input
logic pin12);

module mux2to1(input logic x, y, s, output logic m);
```

6.2 Submitting Part III

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

```
module mux7to1(input logic [2:0] MuxSelect, input logic [6:0] MuxIn, output
logic Out);
```

7 Optional: Running your designs on the FPGA board

Now that you have learn how to use Modelsim to simulate a System Verilog file, you can now program your multiplexer implementations on the FPGA board using the following steps:

1. You are provided with a Quartus template project on your Quercus page that can be found here. Download the De1soc.zip file and extract it in the lab2 folder. The extracted folder should contain a couple of files including the de1soc.qsf file that defines how the switches and LEDs connect to the pins, a de1soc.qpf file that is the quartus project file and a top-level module System Verilog file called `de1soc_top.sv`. You can double click the qpf file to open the Quartus project or open Quartus on your window and click "open project".
2. Add the `mux.sv` file to the folder and instantiate the mux module inside the `de1soc_top` module. Note that the mux module is no longer the top-level module and instead the `de1soc_top` is. For Part II and Part III, change the `u0` instance inside the `mux.sv` file to instantiate the new mux2to1 module or the mux7to1 module. Use the port mappings described in Table 1 and 2.
3. Follow the lab tutorial video instructions to compile and load your program onto the board.
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.

mux2to1 module Port Name	Direction	DE1-SoC/DE10-Lite Pin Name
x	Input	SW[0]
y	Input	SW[1]
s	Input	SW[9]
m	Output	LEDR[0]

Table 1: Mux2to1 port mapping to DE1-SoC/DE10-Lite pin names

mux7to1 module Port Name	Direction	DE1-SoC/DE10-Lite Pin Name
`MuxIn`	Input	SW[6:0]
`MuxSelect`	Input	SW[9:7]
`Out`	Output	LEDR[0]

Table 2: Mux7to1 Module port mapping to DE1-SoC/DE10-Lite pin names