Lab 1: Selector

Lab 0 (https://umich.instructure.com/courses/460347/pages/lab-0-tutorial)	Total Points: 50	Lab 2 (https://umich.instructure.com/courses/460347/pages/lab-2-timing)

Combinational Design I

Overview

The main goal of this experiment is to familiarize you with more features of the DE2-115 board and to introduce additional constructs in Verilog that can greatly simplify design entry. This being your first "real" lab, the actual design work required is quite minimal.

Preparation

1. Read the **Lab Overview document** (https://umich.instructure.com/courses/460347/pages/lab-overview) . This is a good starting point and a useful reference for information that you'll frequently need.
2. Make sure you have completed the **Tutorial** (https://umich.instructure.com/courses/460347/pages/lab-0-tutorial) before starting on this lab. This will save you a lot of time!

Design Specification

The top-level schematic of the circuit you need to design is shown in Figure 1. Named **Selector**, it has two 7-bit inputs connected to two different banks of slider switches (SW[16:10] and SW[6:0]). It also has an input connected to the KEY[3] push-button. It has a 7-bit output connected to the red LEDs (LEDR[6:0]) and a 7-bit output connected to the rightmost 7-segment display (HEX0). Locate all of these inputs and outputs on the DE2 board before you continue.

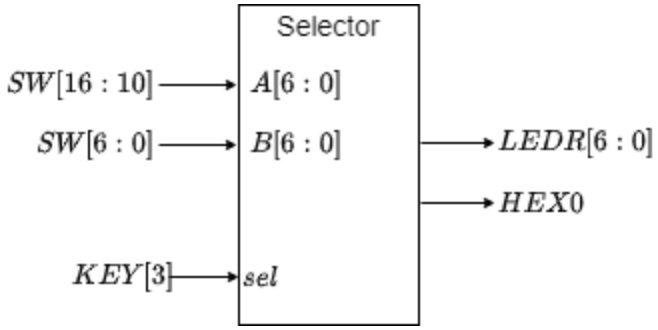


Figure 1: Interface of the Selector Module

Your design should satisfy the following requirements:

- When the KEY[3] push-button is pressed, the 7-bit word specified by the settings of the SW[6:0] switches should be displayed on both LEDR[6:0] and HEX0.
- When the KEY[3] push-button is not pressed, the 7-bit word specified by the settings of the SW[16:10] switches should be displayed on both LEDR[6:0] and HEX0.

You need to describe your design using **Structural Verilog**. We hinted at this style of expressing a design in the introductory **Tutorial (https://umich.instructure.com/courses/460347/pages/lab-0-tutorial)** . Verilog provides built-in modules for all standard n-input logic gates: **and, or, not, nand, nor, xor, xnor**. The syntax for instantiating (i.e., creating an instance of a gate) in your design is:

<gate type> <instance name> (<output name> <input list>)

where <gate type> is one of the above standard gate names (in lower case!), and <instance name> is a unique label that identifies this instance within a Verilog module. The gate inputs and outputs are a comma-separated list within parentheses starting with the name of the gate's output followed by the names of the gate inputs.

As you've seen in the **Tutorial (https://umich.instructure.com/courses/460347/pages/lab-0-tutorial)** , this instantiation mechanism is also available for user-defined modules and allows for a hirerchical design paradigm that greatly enhances readability. A particularly powerful feature is the implicit instantiation of n identical modules that are connected to multi-bit inputs and outputs. For example, assume that we have two 4-bit inputs A[3:0] and B[4:0] and we need to generate the 4-bit output F[3:0] such that F[i] = and(A[i], B[i]) for i = 0, 1 2, and 3. One way to do this is by explicitly instatiating 4 and gates as follows:

```
and g0(F[0], A[0], B[0]);
and g1(F[1], A[1], B[1]);
and g2(F[2], A[2], B[2]);
and g3(F[3], A[3], B[3]);
```

However, a more compact way to achieve the same thing is to *implicitly instantiate an array of 4 and gates*. Assuming A, B, and F are approriately-defined 4-bit signals, the following single

instantiation statement has basically the same effect as the 4 explicit instantiations above :

```
and G[3:0](F, A, B);
```

Figure 2 shows that, other than having different instance names, the circuits produced by explicit and implicit instantiation are identical.

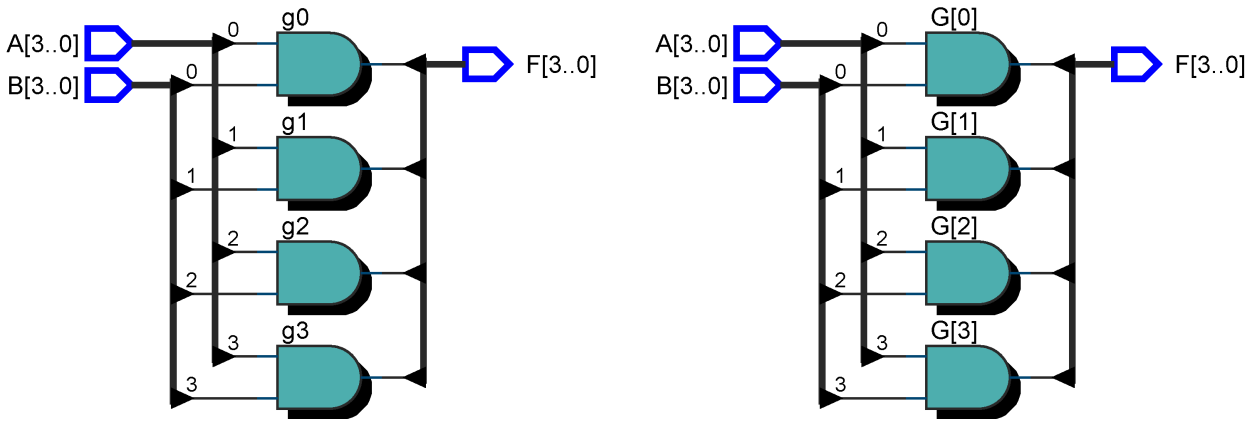


Figure 2: Explicit (left) versus Implicit (right) instantiation of an array of 4 and gates. Explicit instantiation requires that the gates be individually named. Implicit instantiation automatically generates indexed names that correspond to the multi-bit patterns of inputs and outputs.

A variation of implicit instantiation allows for the instantiated modules to share common inputs. For example, module **Single** below describes an *enabled* and gate: when en = 1, f is equal to a, otherwise f is set to 0. Module **Array** shows how you can create 4 copies of this module such that F[i] = and(en, A[i]) for i = 0, 1, 2, and 3, and Figure 3 shows the resulting circuit.

```
1  module Single(en, a, f);
2          input en;
3          input a;
4          output f;
5
6          and s(f, en, a);
7  endmodule
8
9  module Array(en, A, F);
10         input en;
11         input [3:0] A;
12         output [3:0] F;
13
14         Single S[3:0](en, A, F);
15 endmodule
```

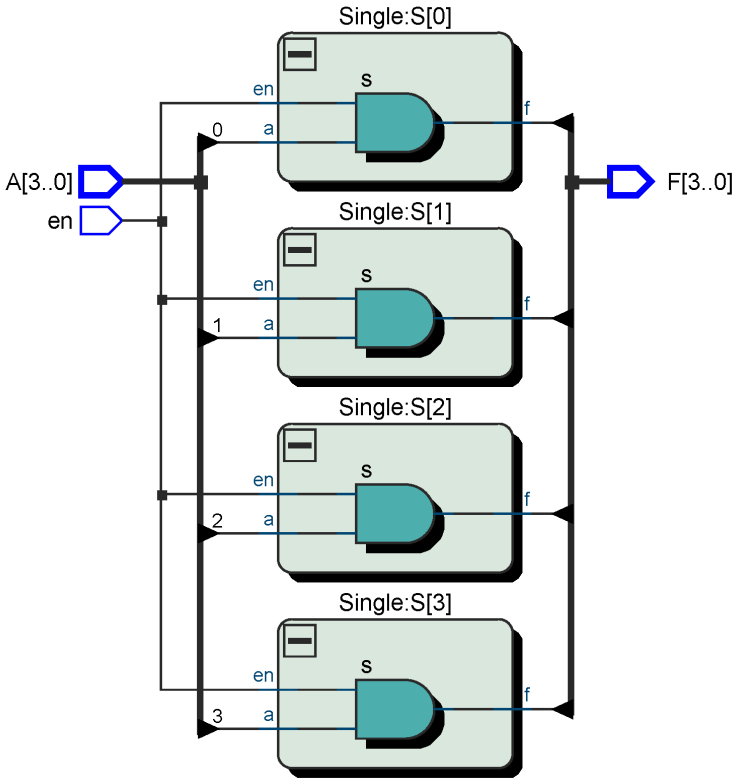


Figure 3: Circuit corresponding to the Array module obtained by implicit instantiation of the Single module

Sytlistic Note: It is good practice to use lower-case names for single-bit variables and upper-case names for multi-bit variables (sometimes called words or bit vectors).

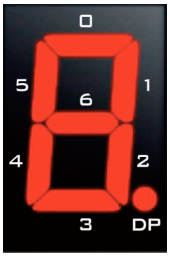
Design Notes and Hints

There a few more things you need to know before you can start.

- Selector is basically a 2-way multiplexer, a circuit that we will study in detail a bit later in the course. Assuming inputs a, b, and sel, and output f, you can think of it as implementing an "if sel then f = a else f = b" statement that you may be familiar with from programming in C or C++. This function can be described in several different ways in hardware. In particular, the following expression for f as a function of sel, a, and b can be easily translated into a **structural gate implementation** that can serve as a bit-level multiplexer module that can be instantiated to create a multi-bit multiplexer:

  f = sel && a || ~sel && b

- The seven segments in the HEX displays are numbered 0 to 6 as shown below. Segment i in HEX0 would be accessed as HEX0[i].



- The HEX outputs are *active low*. That means a HEX segment lights up when a 0 (low) is driven into it. You'll need to think about how to handle that.
- The push-buttons (e.g. KEY[3]) are also active low. This means that the pressed state of the button corresponds to 0 (low) and the unpressed state to 1 (high). This is the opposite of the LEDR outputs and SW inputs which are *active high*.
- Your expected file structure should be as follows:

```
Project Directory: Lab1
Porject Name: Selector
Top-Level Module: Selector.v
Testbench Module: Testbench.v
```

Deliverables

A. Pre-lab (0 points) There is no pre-lab for this lab.

B. In-Lab (15 Points)

Generate the .sof file for *Selector*, download it to the board, and verify that your implementation is working properly. When you are satisfied that you have a correctly-functioning circuit, print the In-Lab certification sheet and demonstrate its operation to a 270 lab instructor. Take a photo of the signed demo sheet and submit it to Gradescope. The instructor will collect the demo sheet.

C. Post-Lab Questions (20 points)

Submit answers to the following questions:

1. Consider a MUX which has a single control line (C) which selects between two inputs (A and B) to generate one output (M). Draw a truth-table for this device. Assume the input A is selected if C=0, otherwise the input B is selected. (5 points)
2. If you were to construct a truth table for *Selector*, how many rows would it have? How many (input and output) columns? (5 points)
3. Assuming that SW[6:0] = 0b101 1110 and SW[16:10] = 0b001 0111, which red LEDs remain lit regardless of whether the KEY[3] is pushed or not? Note: The notation 0b1011110 represents a binary number. We will discuss this in more detail later. For now, you can assume SW[6] = logical 1 or switch on and SW{5}= logical 0 or switch off, etc. (4 points)
4. Assuming KEY[3] is 1, which switches must be set to what values if you want a "7" to be

displayed on the seven-segment display? What if you wanted a "1" and KEY[3] was 0? If a given input doesn't matter, don't list it as part of your answer. (6 points)

D. Post Lab Simulation (15 points)

- Submit the *Selector* Verilog code. (10 points)
- Submit the *Selector*'s simulation waveforms and test bench showing its response on the appropriate 7-segment display and LED outputs to all possible combinations of the SW[0], SW[10] and KEY[3] inputs. You do not have to simulate the other inputs and outputs. You must implement the following truth table for your test procedure. (5 points)

KEY[3]	SW[0]	SW[10]
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1