# LAB 3 –Verilog for Combinational Circuits

## Goals

- Learn how to implement combinational circuits using Verilog.
- Design and implement a simple circuit that controls the 7-segment display to show a 4-bit value in hexadecimal format.

## To Do

- Design a circuit that gets a 4-bit binary value as an input and generates 7 control signals that drive the 7-segment display.
- Use the adder circuit from Lab2 and show its output on the 7-segment display.
- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.
- To complete the lab you have to show your work to an assistant before the deadline, there is nothing to hand in. The required tasks are clearly marked with gray background throughout this document. All other tasks are optional but highly recommended. You can ask the assistants for feedback on the optional tasks.

## Introduction

In Lab 2 exercise, we used the LEDs to display the result of our adder. Instead of showing the result as a binary number, we could represent the number in a more human readable format using the 7-segment display on the Basys 3 board. A 7-segment display consists of seven separate LEDs in a single package. Each of the seven segments is labeled using the letters a, b, c, d, e, f, g (see Figure 1). We can use the 7-segment display to represent different characters or digits by making particular segments glow at the same time. In this lab, we will implement a circuit that could show the hexadecimal characters as shown in Figure 2. For example, if we want to display 0 on the 7-segment display, our circuit should make sure that all of the LEDs except 'g' glow.

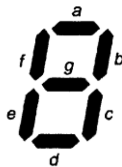


Figure 1: 7-segment display

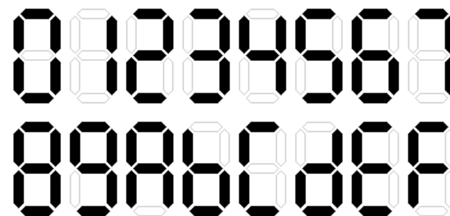

Figure 2: Hexadecimal characters represented in a 7-segment display

Each one of the seven segments has a corresponding pin connected to the FPGA, just like the LEDs in Lab 2. In our circuit, we will need to convert a 4-bit binary input to drive the correct 7-bit signals. One thing to note is that, unlike the LEDs which glow when a logic-1 is applied, the segment in our FPGA board glow when a logic-0 is applied. As we have seen in the lecture, such signals are called active-low. You will notice that there are a total of four 7-segment displays on the FPGA board. To save on the number of pins connecting to the FPGA chip, all four 7-segment

displays share the same inputs. To select which of the four display will act on the inputs, there are four additional "activation" inputs. If we activate all displays at the same time, all of them show the same number, i.e., the same segments glow on all displays. If we want to show different characters on each 7-segment display, then we need to activate the displays separately over multiple cycles. For example, our circuit should first activate the first display and drive the number to be displayed as an input. In the next cycle, our circuit can drive a different input by selecting another 7-segment display. Since the human eye can only notice changes slower than approximately 20ms, if you do this fast enough (more than 50 times per second) you will see a stable display. This is the same trick used in movies and television sets. In this exercise, we do not implement this trick as we do not implement sequential circuits in this lab. Instead, you will show the same character on all of the 7-segment displays. In the report, you will turn three of them off and just use one of the 7-segment displays to show the number.

In this lab, we will reuse our adder circuit from Lab 2. Our goal is to show the results of the adder on the 7-segment display. However, the output of the adder is 5 bits, we cannot show the results as a single hexadecimal character in all cases. Therefore, we will continue using an LED to represent addition results higher than 15. Basically, in your design, the most significant bit of the addition should be connected to an LED and the rest of the bits should drive the 7-segment display.

## Part 1- Converting a Binary Number to 7-Segment Display Encoding

As a first step, fill in the truth table below that converts a binary number to a 7-segment encoding. Note that a segment should glow when the corresponding output is logic-0.

| Display | S3 | S2 | S1 | S0 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | | | | | | | |
| 2 | 0 | 0 | 1 | 0 | | | | | | | |
| 3 | 0 | 0 | 1 | 1 | | | | | | | |
| 4 | 0 | 1 | 0 | 0 | | | | | | | |
| 5 | 0 | 1 | 0 | 1 | | | | | | | |
| 6 | 0 | 1 | 1 | 0 | | | | | | | |
| 7 | 0 | 1 | 1 | 1 | | | | | | | |
| 8 | 1 | 0 | 0 | 0 | | | | | | | |
| 9 | 1 | 0 | 0 | 1 | | | | | | | |
| A | 1 | 0 | 1 | 0 | | | | | | | |
| B | 1 | 0 | 1 | 1 | | | | | | | |
| C | 1 | 1 | 0 | 0 | | | | | | | |
| D | 1 | 1 | 0 | 1 | | | | | | | |
| E | 1 | 1 | 1 | 0 | | | | | | | |
| F | 1 | 1 | 1 | 1 | | | | | | | |

The truth table you have filled defines a circuit with 7 outputs. If you wanted, you could write the Boolean equations for all outputs separately, try to find out a (somewhat) optimized version of the equations, see if you can share portions of the gates among the different outputs (reducing the number of gates) and then draw the schematic for the resulting Boolean equations.

## Part 2 - Implementing a Circuit to drive the 7-Segment Display

Before we continue, recall the "bus" concept that we know from the lectures and Lab 2. For the 7-segment display, we have seven outputs that we have named A to G. Instead of making seven separate connections, we can combine all of them together in a single bus. Assume that you call this bus D[6:0]. An important decision is how the bus signals should be connected to the individual outputs that drive the 7-segment display. Is A connected to D[0] or to D[6] (or to another signal)? You are free to choose what kind of assignment you want to make to the buses.

Start with opening your Lab 2 project on Vivado. *You may want to create a copy of the Lab 2 project if you would like to keep the current version of it.* Inside Flow Navigator, under "Project Manager", click on "Add Sources". Select "Add or Create Design Sources", and click next. Click "Create File". A new dialog box will pop up. After you choose a name for the module (e.g., 'Decoder') clicking on the 'Finish' button will open a new window where you can easily add the connections. You can type in the name, select the direction (i.e., input or output), and you can specify the range of buses. You need to define 4-bit input and 7-bit output signals for your module that will convert a binary number to a 7-segment display encoding.

Now you are ready to describe the functionality of your new module using Verilog. You can try different styles to see which one suits you well. It is important to realize that the length of the code is not proportional to the complexity of the hardware. Try to write well-structured code using simple statements and document them well.

Once you have created the file, you can start writing Verilog code to implement the functionality of your new module. Basically, you need to implement a logic that will make the corresponding segment glow for each input number.

Note that in case you have a procedural assignment to a wire (inside an 'always' block) you should make sure that it is declared as 'reg'. This is also the case for outputs that are defined in the module declaration. In general, all signals on the left-hand side of <= or = in an 'always' statement must be declared as 'reg'.

In case there is any syntax error in your code, you can see a red underline along with a red marker on the scrollbar. Move your mouse to the scroll bar and you can see the cause of the error. Additionally, you can go to Window → Messages which also shows all the errors (including the warnings) in your code. ***You may have to save the file before the messages are updated.***

## Part 3 – Showing the Addition Result on the 7-Segment Display

Our goal is to show the result of our adder circuit from Lab 2 using the 7-segment display. To achieve that, we need to attach an instance of the new module that we just implemented (we will just call it "Decoder" from now on) to the output of the adder, i.e., the output of the adder should be an input to the instance of the Decoder. To do that, we need to create a new "top" module that will create an instance of each module and make appropriate connections between them.

Create a new source file and instantiate the adder and the Decoder. In this module we will still have the two inputs A[3:0] and B[3:0] that go directly to the adder. The output of the adder is a bus that is 5-bits wide. 4 of these S[3:0] will go to the decoder, and the most significant bit S[4] should connect to a separate LED. An example schematic is given in Figure 3.

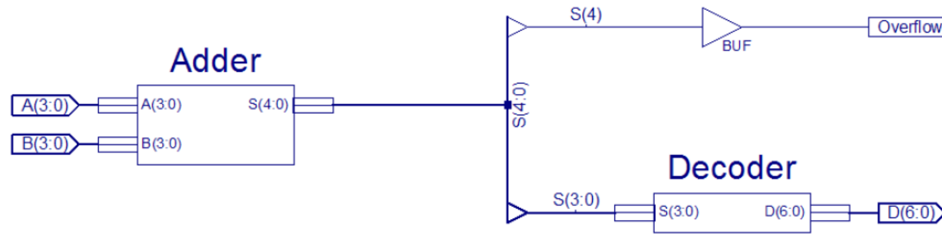

Figure 1: An example top-level schematic

Connect the output of the adder to the input of the Decoder as you see appropriate. Make sure that you do not have any errors.

In case you would like to see the schematic representation of your code, go to Flow -> Elaborated Design. You can ignore the warning dialog.

We also need to modify the constraints file (the .xdc file) that tells us how to map the outputs of our top module to the FPGA pins. Note that in your top module you should have seven bits going to the 7-segment display and one bit connecting to the LED.

We assume that you call the output D[6:0] and A is D[0], B is D[1], C is D[2] etc. For the inputs, assume that we have a 4-bit bus called s[3:0].

Find and open the .xdc file that you created last week for Lab 2. Make necessary changes to select the pins connecting to the 7-segment display. We provide example constraints below assuming that you would like to connect D[0] to segment A, D[1], to segment B and so on. Also, make sure to update the constraints for the inputs if you use different input names in your new top module.

Example connections for the 7-segment display are the following:

```
set_property PACKAGE_PIN W7 [get_ports {D[0]}]
set_property PACKAGE_PIN W6 [get_ports {D[1]}]
set_property PACKAGE_PIN U8 [get_ports {D[2]}]
set_property PACKAGE_PIN V8 [get_ports {D[3]}]
set_property PACKAGE_PIN U5 [get_ports {D[4]}]
set_property PACKAGE_PIN V5 [get_ports {D[5]}]
set_property PACKAGE_PIN U7 [get_ports {D[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {D}]
```

Now we just have to generate the programming file of the entire project and download it to the FPGA. Then we can finally check the result using the 7-segment display.

Using Lab 2 as a reference, generate the programming file and program the FPGA. Show the working circuit to an assistant.

## Last Words

In this exercise, we deliberately left some options (how to add the display decoder) up to you. There are frequently many ways a specific task can be completed, and it is not immediately clear which of the alternatives is the best choice. More often than not, there is not really a ‘best’ option; all choices would work more or less equally well.

While working on your implementation, you may come across various problems. Here are some

tips to debug your circuit implementation.

- Isolate the problem. You can just map your new Decoder module to the FPGA board by connecting 4 inputs to the switches and the 7 outputs to the 7-segment display. This way, you can check if the Decoder works correctly. If this is the case, the problem could be in the adder circuit or in the interconnection between the two.
- If something is displayed on the 7-segment display, but it does not match the expectations, consider that all output LEDs are independent. Try to find out which LEDs function correctly (a, b, c, etc.) and which ones do not. You should then find the place in the Verilog code that determines the output of the LEDs that are not working correctly.
- Check your bit ordering. The given constraints for port D are only valid if you followed the example ordering of bits (i.e., D[0] corresponds to segment A, D[1] to segment B, etc.)

Until now, we have designed combinational circuits. The outputs of these circuits were directly determined by their inputs. We use these circuits extensively, but they are not able to remember what has happened in the past: they have no memory, only the present time. Starting with the next exercise, we examine state holding circuits that can differentiate among different states and can move through different states depending on the input and the present state. These circuits are called finite state machines.