ECE333 - Digital Systems Lab

– Lab Report –

Lab 1 - 7-Segment Display Driver

submitted by

Liaskonis Spyridon

November 13, 2022

Liaskonis Spyridon
sliaskonis@uth.gr
Student ID: 03381

Lab Report Summary

The following report is divided into six sections. The first section is an introduction to the aim of this project. The sections two, three, four and five are more detailed paragraphs about each part of the assignment. They focus on the implementation of each part of the 7-Segment Display driver, how we verified the implementation and also the experiments that were conducted on the FPGAs. The last section is the conclusions in which we talk about the progress of the assignment in fields such as design, verification and testing as well as difficulties that may occurred and how we overcame them.

Contents

1 Introduction

The aim of this project is the implementation of a 7-Segment display driver for the Nexys A7 board. The implementation is divided into 4 parts each one developed separately and with separate test-files/testbenches. Each part focuses on a different objective, however (almost) all the parts are needed for the full implementation of the LED driver. The objectives for each part are the following:

- Part A: Implementation of the LED decoder.
- Part B: Driving four digits to the LED display.
- Part C: Incremental rotation of a message on the display using one of board's buttons.
- Part D: Incremental rotation of a message on the display with constant delay.

All of the objectives above were successfully implemented. In particular, here's a quick summary of the implementation of each part:

- Part A: The LED decoder was implemented using a simple decoder having as input the character we want to display and output the value of each segment.
- Part B: The driving of the four digits was implemented using a counter responsible for driving each of the 4 anodes of the display to 0 without them overlapping and also fetching the data of each digit at least 0.20 µs before their anode was active (0). More about the time that got chosen to fetch the data (and why that time was chosen) is talked in section 3.
- Part C: The rotation of the message with the use of a button was implemented again by using a counter, responsible for the shift of the message by one place to the left.
- Part D: Last, the rotation of the message with a constant delay was implemented by using the same counter used in Part C, only this time this counter was "controlled" by another counter. The second counter here was used for the creation of a delay and acted as an enable signal to the first counter.

2 Part A - Implementation of the LED Decoder

2.1 Implementation

The LED decoder was implemented using a combinational circuit consisting of just a decoder. Here is the dataflow of the first part:

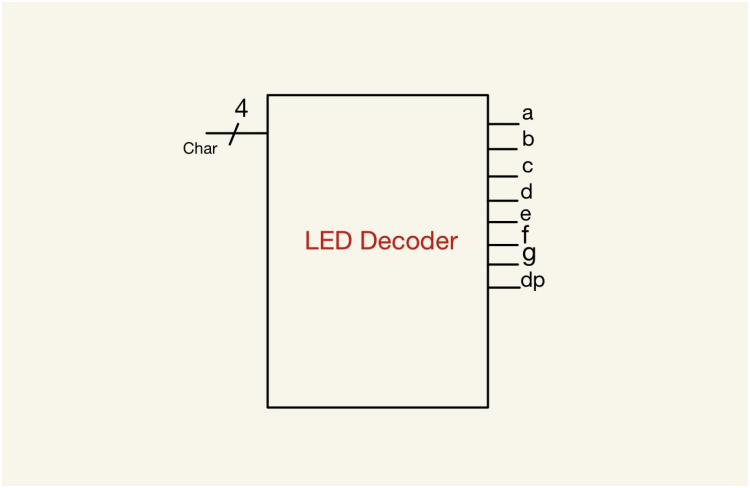

Figure 1: LED Decoder dataflow

The LED Decoder circuit consists of just one always block including the decoder's implementation. The input of the decoder is a 4-bit value called char, representing the character that we want to decode and later display. The output of the block is controlled with a case statement that outputs all of the segments's values (a, b, c, d, e, f, g, dp) according to the character inputted. Notice that in the current implementation the output of the decoder consists of all the 7 segment signals plus the decimal point signal that is always set to high since the display of floating point numbers or any other characters that use the decimal point is not needed in the context of this assignment.

2.2 Verification of implementation

The verification of the implementation of part A was done by inputting all possible combinations of the 4-bit char value (from 0000 to 1111) and checking if the output for each segment matched the output expected. In particular, the testbench consisted of the instantiation of the LED decoder module and an initial block in

which every 100ns we changed the char's value starting from 0 (binary: 0000) up to f (binary: 1111). The percentage of coverage of the control vectors used for the verification in the current state was 100% since all the values of the hexadecimal system were inputted and checked.

∗The message that we aim to display on the 7 segment display is the following "123456789abcdef". Thus, that's why the percentage of coverage is 100%"

3 Part B - Driving 4 digits to the display

3.1 Implementation

For the implementation of part B we use 4 different modules. These modules are:

- LED Decoder (module designed in part A)
- MMCM (Mixed-Mode Clock Manager)
- LED Driver
- 4-bit Counter

Here is the dataflow showing how all the modules are connected:

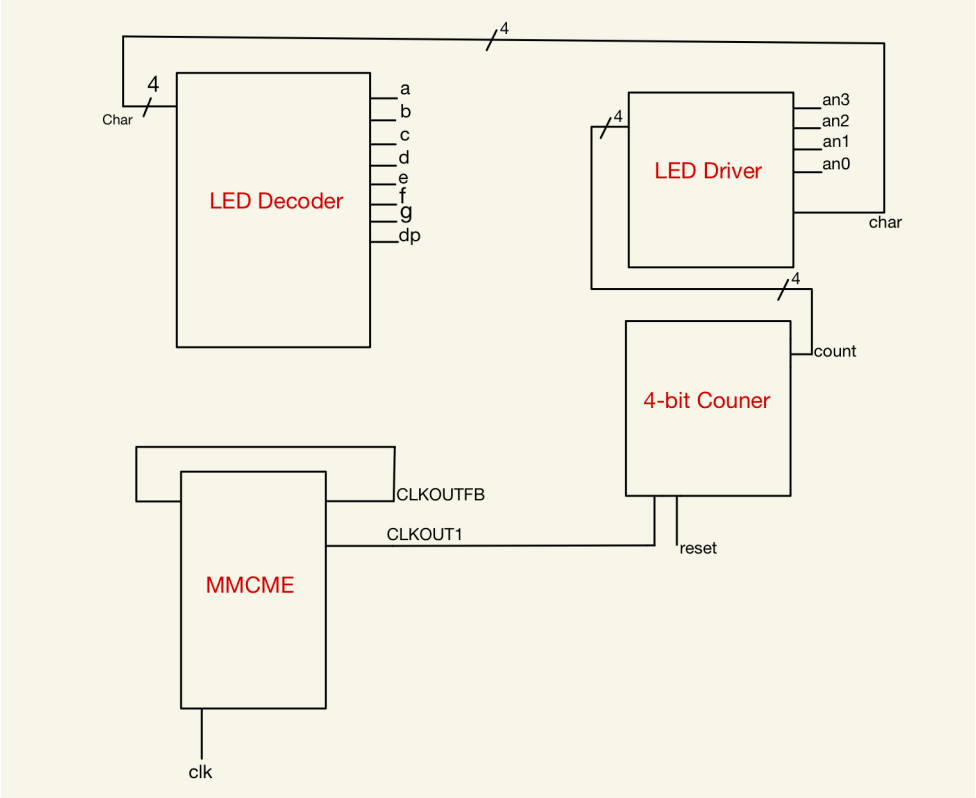

Figure 2: Part-B dataflow

All of the modules shown in the figure above are needed in order for our circuit to work properly. Starting with the MMCM, it's a module that's used for the multiplication of our board's clock period. We multiply the clock period by 20 times and we then get a new clock (CLKOUT1) with a period of 0.20 µs. The reason we need the new clock is the proper charge of each of the 4 digits on the display. Then moving onto the 4-bit counter, it's a module needed for the driving of each anode (an3-0) to 0 without them overlapping each other. This module has two inputs, a clock and a reset signal, and one output, the count. The counter is a sequential circuit consisting of an always block that controls count's values. In particular the counter is initialized to 1111 (using the reset signal) and then it's value gets decreased by one (0001) at each posedge clock signal. Since the counter is a rotational counter when it's value reaches 0000 it starts over at 1111. Each of the counter's value is then an input to the LED Driver module that's responsible for driving all 4 of display's anodes to 0 and also sending the char value (that we want to display) to the LED Decoder two steps before the anode in which it will get displayed is set to

0 (so that the digit on the display gets properly charged). Both of the procedures above, are implemented using two combinational circuits. In particular, two always blocks with one case statement each, that's controlled by count's value.

∗In part B a message is constantly gonna get displayed on the display. That message is the 4 digits "0123". But here instead of inputting the message on the LED Driver module and then initializing the message with a value inside the testbench, we directly assign it to the char value based on the counter's values.

3.2 Verification of implementation

For the verification of the part B, a simple testbench was used consisting only of the instantiation of the FourDigitLedDriver (the top module in which all other 4 modules were instantiated), one always block that was used to generate a clock signal with a period of 10ns and an initial block in which we start the clock and also set the reset signal to 1 (so that the circuit gets initialized and running). We then checked to see if the anodes were correctly driven to 0 and also if the char value was decoded two steps before an anode was set to 0 by observing the waveforms. Here is the result of the testbench:

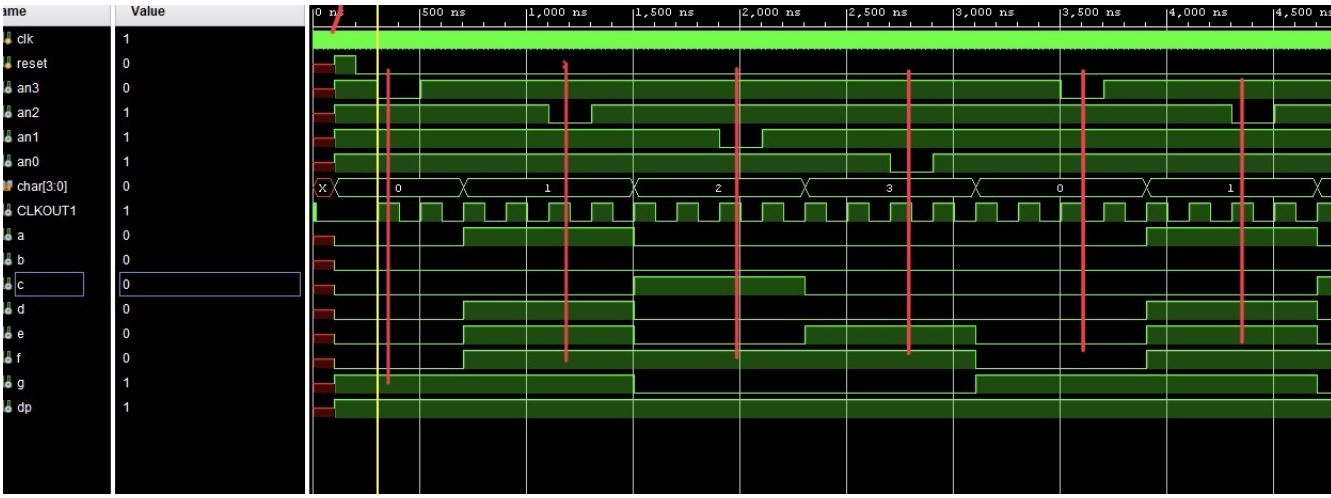

Figure 3: Part-B, Testbench waveforms

3.3 Experiment and Final Implementation

The programming of the FPGA for the part B was pretty straight forward and no problems occurred. The message got displayed on the display successfully after the first testing. No electrical problems occurred as well and no changes happened to the code happened for that part.

4 Part C - Incremental rotation of the message using a button

4.1 Implementation

For the implementation of the part C, six modules were used. Specifically, two new modules were added and connected with the modules from part B and one of the previous modules was slightly altered. Here is the dataflow for the current part:

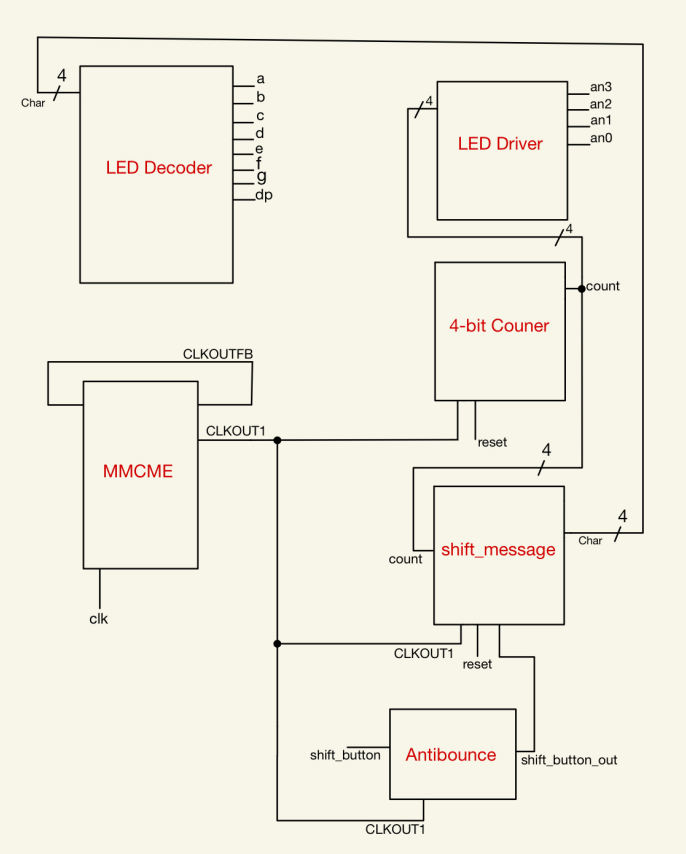

Figure 4: Part-C dataflow

In the current implementation of the driver all modules from the previous parts stayed the same except the LED Driver module, which now consists only of one combinational circuit - one always block that's responsible for driving the four anodes to 0.

Now, for the part C, a button was needed in order for the message to rotate, thus an antibounce circuit was necessary. The antibounce circuit would be responsible for

handling unpredictable bounce in the signal of the button when it was toggled. The antibounce module shown above does this exact job using three D-flip flops as filters. The filtered signal is then used on the shift_message module. On this module we have four always blocks that make three combinational and one sequential circuits. Starting off with the first always block, its a sequential circuit implementing a 4-bit counter with asynchronous reset and an enable signal which in this case is the shift_button_out signal. Here we want the counter to only get increased whenever the button is pressed, thus that's why we use the button signal as an enable signal. Next, the value of the counter is used in the second always block, in order to calculate the address of the "active" memory, the memory that contains the value of the digit that's gonna get displayed each time. Since four digits will get displayed each time, 4 different calculations for the address are needed. In particular the address of each digit will get calculated as:

```
First digit: address = count2 + 4'b0000;
Second digit: address = count2 + 4'b0001;
Third digit: address = count2 + 4'b0010;
Fourth digit: address = count2 + 4'b0011;
```

Then using the address calculated above, a value is assigned to char from the memory. The assignment is done in the third always block and the memory is initialized in the fourth always block of the module. Just like part B, we should know the value of the char, two steps before the anode that it will get displayed at is set to 0. In this case thought, the char has 16 possible values that come from the memory message[]. So that's why we need to calculate 4 different addresses, one for each digit and since the characters that will get displayed from the memory are all next to each other, that's why the address takes values from $[count2, count + 4]$. Last, count1 (4-bit counter's output) is used here to calculate the address two steps before the anodes are set to 0.

4.2 Verification of implementation

For the verification of part C we used a simple testbench again, consisting of the instantiation of the top level module (FourDigitLedDriver), one always block responsible for the generation of the clock and an initial block in which we start the clock and set the reset signal to 1 (so that the circuit gets initialized and running) and then back to 0. Also in the initial block we set the shift_button to 1 and then

back to 0. The shift_button signal stays in each state for 3000 time units and it swap between those stages 15 times, enough for a whole rotation of the message. We then check to see if the message rotated successfully by observing the waveforms. Here are some results of the testbench:

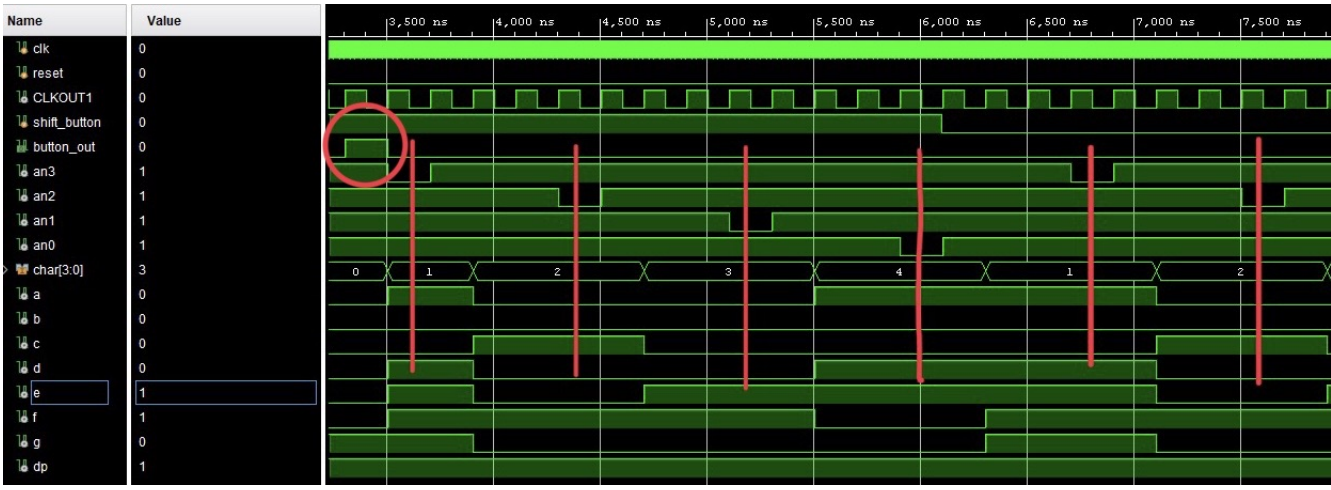

Figure 5: Part-C, Testbench waveforms

Figure 6: Part-C, Testbench waveforms

In figure 5, we see how after the button_out signal is set to high the whole message gets shifted one place to the left. From displaying the message "0123" we now display the message "1234". In figure 6 we see how the rotation of the message works. We press the button at the end of the message and we notice that the message goes from "cdef" to "def0".

4.3 Experiment and final implementation

The programming of the FPGA for the part C happened around 5-6 times cause of faulty code. There was a problem with the rotation of the message in the current code so that's why i tried to come up with a new way to shift the message. However the new way i came up with caused many other problems like 1 digit appearing accross the whole display (actually across both the displays even thought i had only programmed the anodes for one of them). At the end i ended up fixing the problem with the rotation of the message on the first code i had written and everthing worked well while programming and testing the FPGA.

∗The problem could've been known even from the simulation however i only noticed it when using the FPGA

5 Part D - Incremental rotation of the message with constant delay

5.1 Implementation

For the implementation of the part D again six modules were used. Five of the modules were the same as Part C, however in this part the antibounce module was not needed since no buttons were used. Its place got a new module called "delay_circuit" that was responsible for generating a 1.6sec delay for each digit to move one place to the left. Here is the dataflow for the part D:

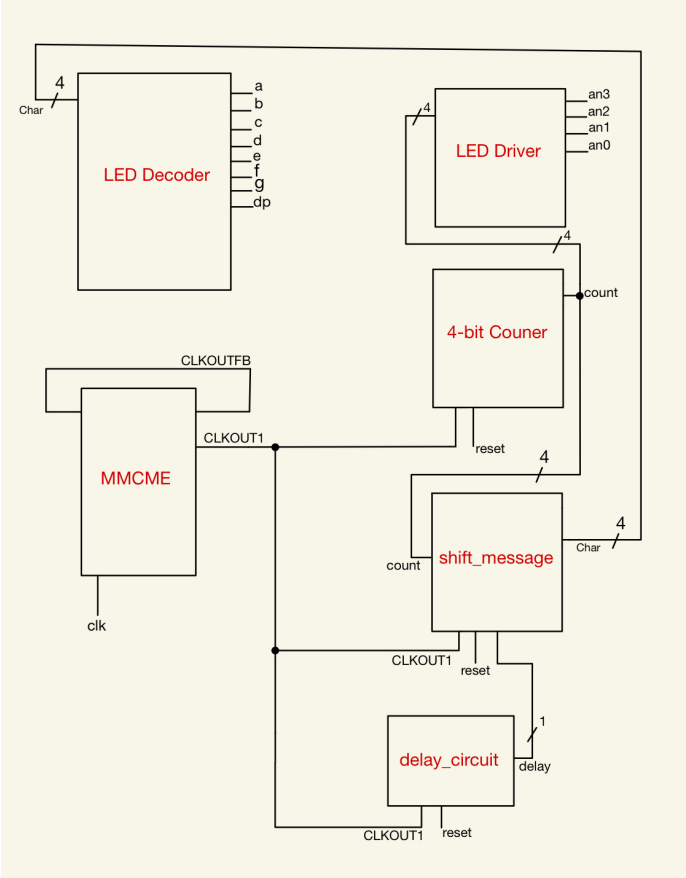

Figure 7: Part-D dataflow

Part D is very similar to the previous part so not much will be talked about for the modules shown above. The only module we will talk about in this part is the delay_cirtcuit module. The delay module has two inputs CLKOUT1 and reset, and one output which is a 1-bit signal called delay. It consists of one always block which

implements a sequential circuit. The delay signal acts as an enable signal to the shift_message module. Basically it does what the shift_button_out did in part C. The 1.6sec delay here is achieved by using a 23-bit counter. The counter gets increased by 1 at each posedge clock signal. When the counter reaches its maximum value the delay signal is set to 1 meaning that 1.6s have passed. When that happens, the counter implemented in the shift_message module gets increased by one meaning that the addresses of the four digits that get displayed are gonna get changed and thus all of the digits are gonna move on place to the left.

5.2 Verification of implementation

For the verification of part D we use the same testbench as part C with the only difference being that we remove all the shift_button signals from the testbench. So that makes the test file consist of just the instantiation of the top level module, the generation of the clock and an initial block where we start the clock and also set the reset to 1 (and then back to 0). However in order to check our driver in this part, we have to make some changes to the delay module. We have to make the counter from 23-bit to something smaller so that we can actually observe how the message rotate on the screen. We then check to see if the message rotated successfully by observing the waveforms. Here are some results of the testbench:

Figure 8: Part-C, Testbench waveforms

5.3 Experiment and Final Implementation

The programming of the FPGA for the part D went also pretty well since not much changed from part C. The message got display and rotated with the expected delay. No electrical problems occurred and no changes happened to the code for that part.

6 Conclusions

The purpose of the assignment was the implementation of a 7-Segment Display Driver. The progress of the implementation of the driver was divided in four different parts. First, we had to begin by building smaller blocks such as the LED decoder, the 4-bit counter and the LED driver that we later connected all together in order to do the first important step that was the driving of 4 digits to the display. After that we had to make the digits displayed on the display move with the push of a button or after a certain amount of time by applying a constant delay. That was implemented as well by using the modules mentioned above combined with some new ones like the shift_message, that used a 4-bit counter in order to move the message one place to the left. However in order for the shift_message module to work we also needed two more modules, different for each part. The antibounce and the delay circuits. Both acted as an enable signal to the shift_message module and were used in order to rotate the message. In terms of testing we then used different testbenches for each part and observed our circuit's behaviour through its waveforms. Most of the problems occurred had to do with bugs on the code but they all got fixed after debugging using the testbenches.