HY430 – Digital Systems Lab

Lab assignment 1

30/10/2019

Βαγενάς Αναστάσης

This report describes the implementation of the 1st laboratory work where it is divided into 4 parts and in each of one them we show the implementation in code/ logic, the verification of its operation and the results of the experiment on FPGA.

Prelude

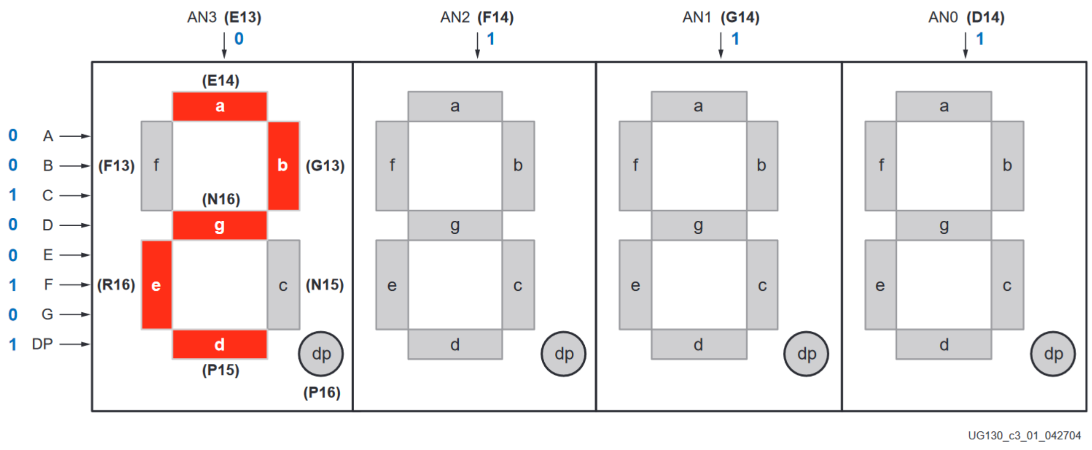
The aim of the laboratory exercise is the implementation of a driver for the four 7-part LED indications of the Spartan 3 board and a rotating presentation of the message of 16 characters.

Part A

Implementation/Verication

In part A we are called to implement a decoder that will have as input the character we want to show and as output the appropriate combination of signals to activate the correct parts of the screen.

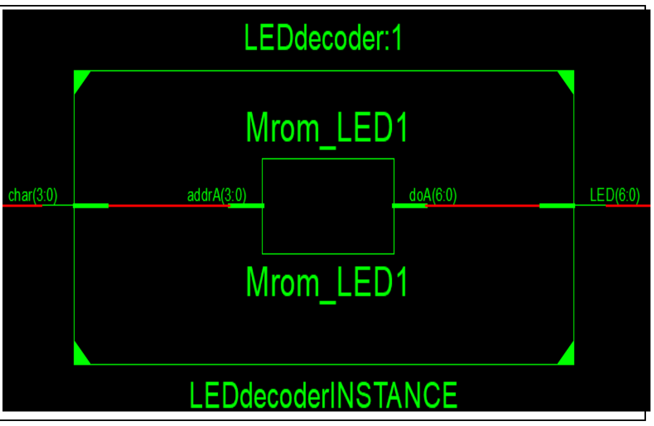
Below is a schematic of the signals we handle (namely the signals A, B, C, D, E, F, G, DP):



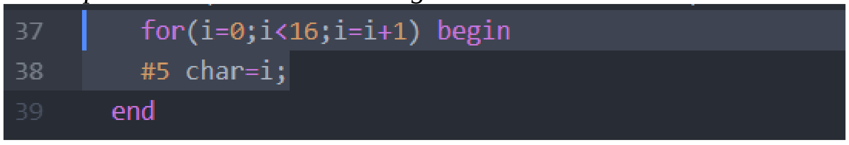
So, for example, if we want to display the character ΄1΄ then we have to drive the signals B, C to logical 0 and the rest to logical 1 to turn on the lights on the screen. Respectively we will do appropriate for the other 15 characters that we want to portray.

At the code level, to achieve this, a case structure is used in Verilog and we assign the LED signal value based on the character value. In addition, since it is a combination circuit, the meaning of the clock is nowhere to be found, except for the CHAR value which is the input to the module..

```
module LEDdecoder(input [3:0]char,output reg[6:0] LED);
always @(char) begin
  case(char)
    4'd0:LED = char_0;
          …
          …
end
endmodule
```

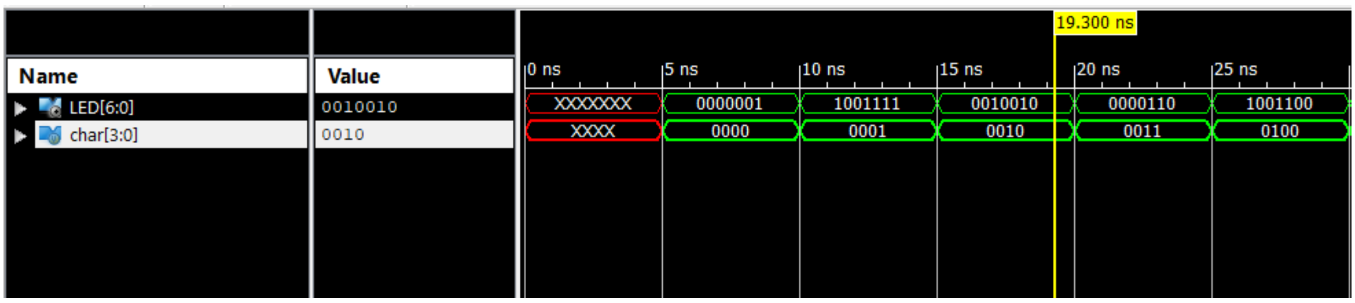


The implementation code in Verilog and its module schematic



```
    for(i=0;i<16;i=i+1) begin
    #5 char=i;
  end
```

Test code



```
Name        Value
LED[6:0]    0010010
char[3:0]   0010
```

Above is a snapshot of the testbench, where in the specific snapshot the value of the character is 0010 or '2' and we have as output the 0010010 which is correct. We do the same for the other characters.

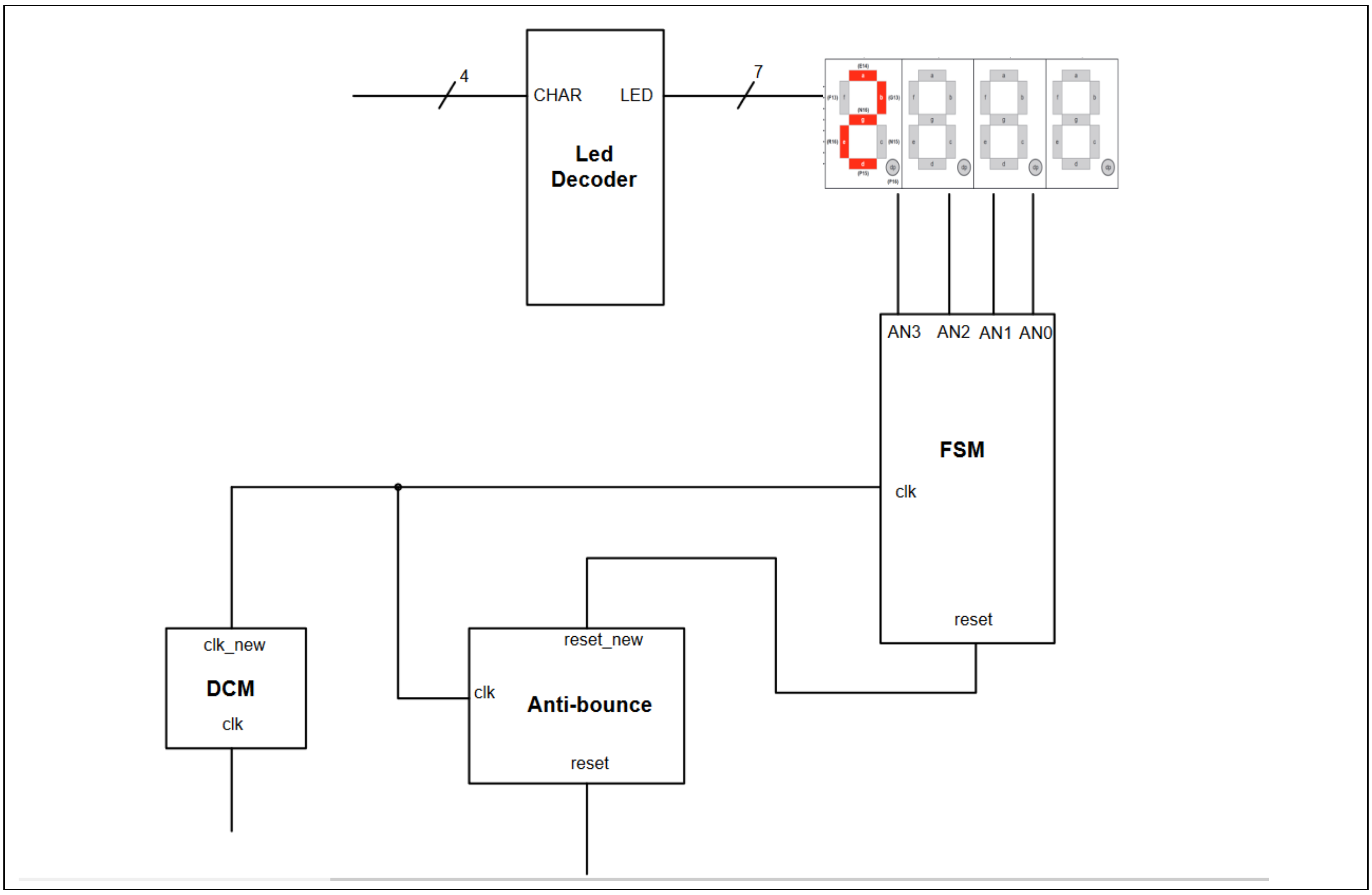
In this part no simulation was needed on the board so there is no experiment.

Part B

Implementation

In part B the skeleton of the laboratory exercise is implemented where we drive the clock through a DCM (Digital Clock Manager) to get a multiple of it, we drive the anodes ANx properly through a counter/FSM and create a button debounce circuit and connect them appropriately.

A schematic of the connectivity down below:



Initially DCM takes the FPGA clock *clk* Initially DCM takes the FPGA clk clock and divides it by 16 and distributes it to the underlying modules of the circuit as *clk_new* (FSM only for now).

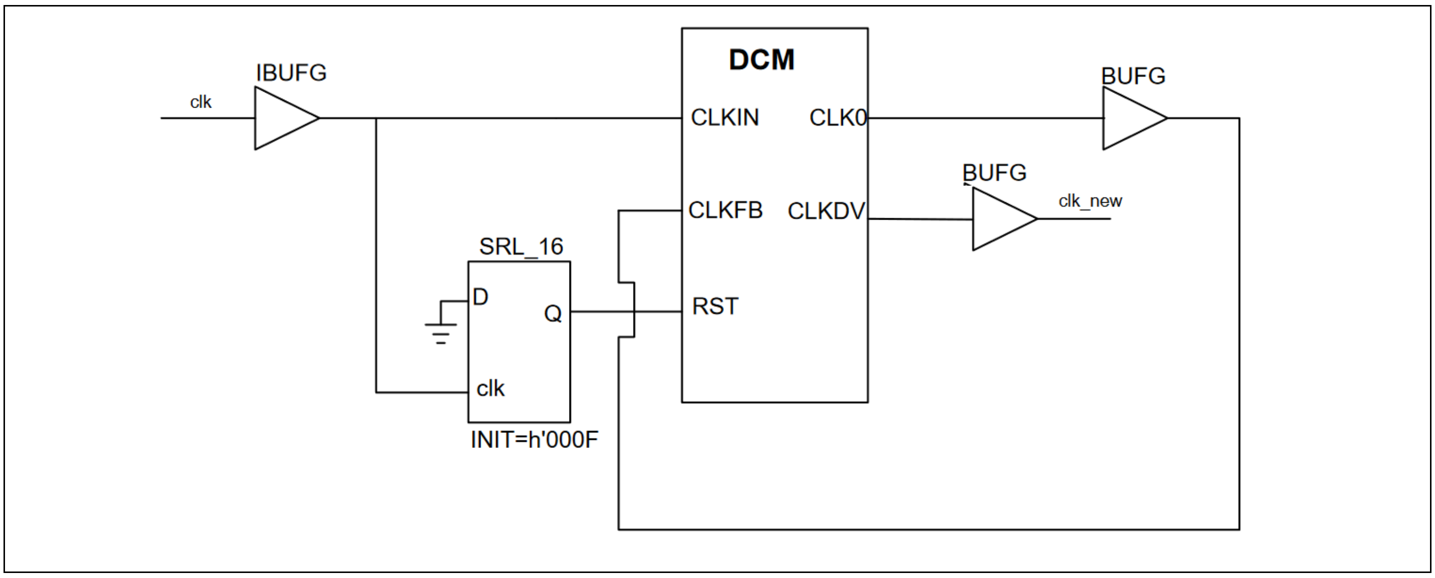
The anti-bounce module on its part takes the reset signal that we have on a button in the FPGA, synchronizes it with the clock and filters the frequent bounces of the signal since it is a mechanical button. In that way *reset_new* then is the new circuit reset and is used in the other modules (FSM only for the time being).

The FSM is the finite state machine (simple counter) required for laboratory exercise that drives the anodes properly so that the parts of the screen light up properly..

Finally, LedDecoder is the module from part A and its function remains the same.

DCM

Below is the connectivity of the DCM module:



As shown, we pass the input-output signals through buffers (recommended based on the DCM manual). So each signal leaving and entering the DCM passes through the appropriate buffer (IBUFG if it is an input signal or BUFG if it is an output signal or internal signal).

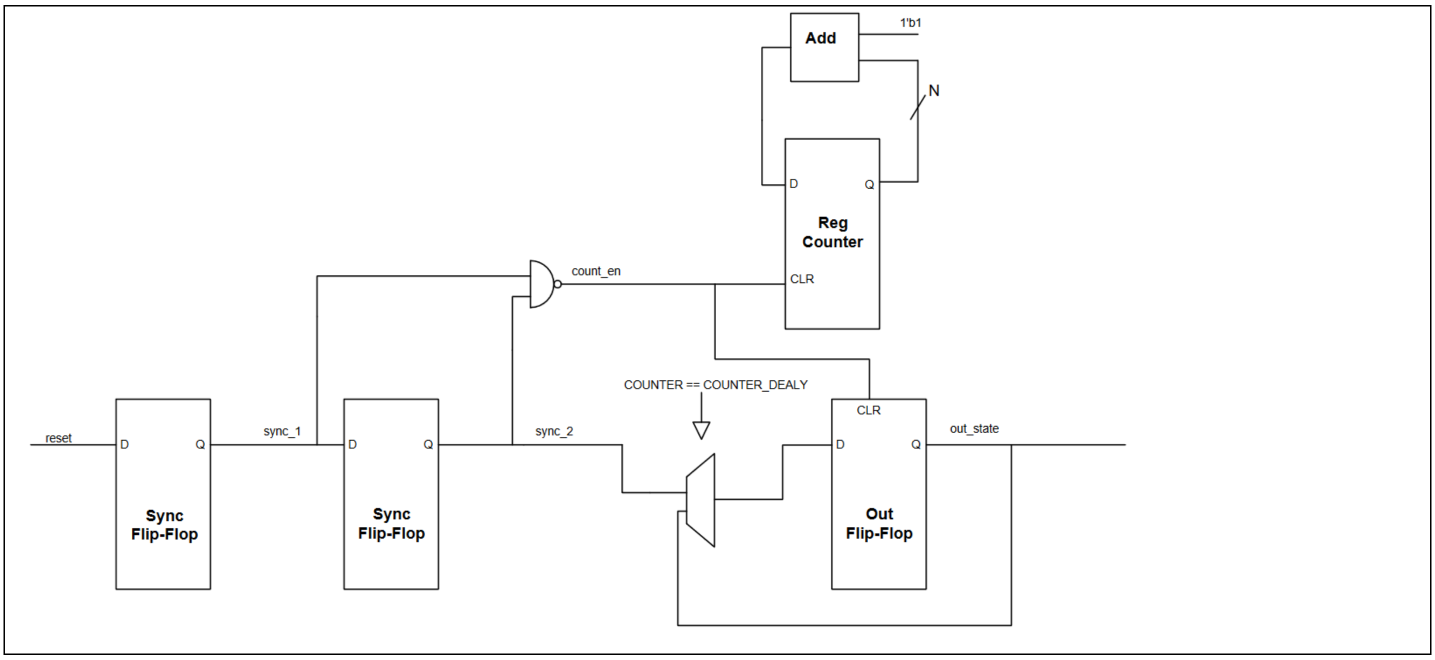
For the smooth operation of the DCM we must have CLK0 clock feedback on the CLKFB so that clock is stable over time.

Also, the rst we give to DCM is applied by a 16-bit shift register (SRL_16 - again primitive by Xillinx) initialized to h’000F which will give us an rst pulse for 4 clock cycles, enough to lock DCM in proper operation.

All this wiring is in a separate module called dcm_manage, where we instatiate all the primitives and interconnect them properly.

Anti-bounce

Below the connection of Anti-bounce module:



To be able to use an asynchronous reset from the FPGA button we must first synchronize the signal with the clock, for this we deal with any metastability with the 2 input flip-flops. Then if the signals *sync_1* και *sync_2* become ΄1΄ then it means we receive *reset* from the board.

In order to be able to filter the mechanical bounces, we use a counter that will ignore the change to ΄1΄ of the reset for a certain number of cycles and only when it reaches the maximum we pass sync_2 to out_state and we have given the new reset_new to the rest of the circuit. maximum value of the counter is 32,767 (COUNTER_DELAY) which is equivalent to approximately 10.4ms (32,767 x 320 ns), enough time to get out of the button bounces.

As long as the reset remains constant at ΄1΄ from there on, it will continuously switch to the output of the circuit (at out_state) ignoring the value of the counter.

Below is a piece of code:

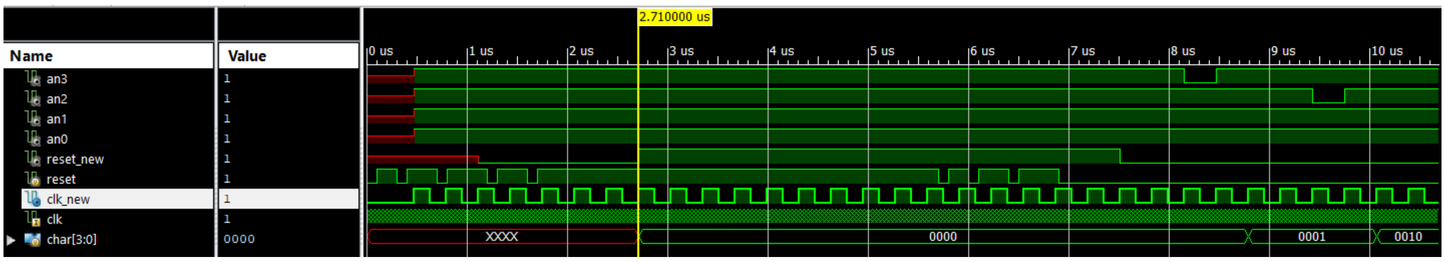
```
module anti_bounce_reset(input clk, in_bit, output reg out_state);
.......
assign count_en = (sync_1 && sync_2);
always @(posedge clk) begin
  if(count_en == 1'b0) begin  //signal not stable yet or 0
             out_state<=0;
              counter<=0;
            end
  else if (counter == COUNTER_DELAY) begin  //counter maxed =>output signal
             out_state<=sync_2;
              counter<=0;
            end
  else
    counter <= counter+1;   //signal stable counter not filled yet
.......
endmodule
```

The code describes the basic anti-bounce function of the figure.

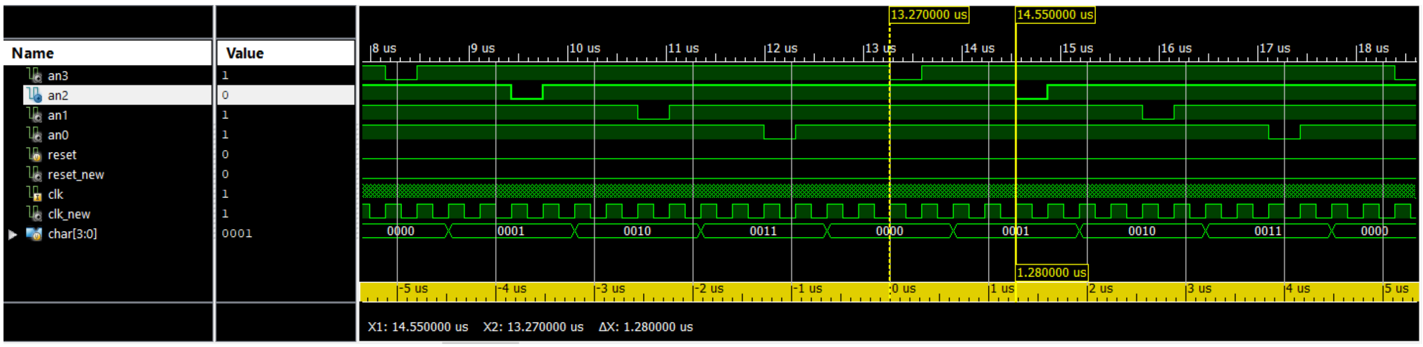
Verification

For verification, again a suitable testbench was used which gives reset pulses, initializes the clock and we just control the output of the circuit in time.

For the anti_bounce_reset module we have set a maximum delay of 3 cycles so that we can see the change in time.



In the simulation image it seems that reset_new ignores the bounce pieces and only after the reset signal is stabilized, after some cycles we will have activation in reset_new (in the yellow cursor).



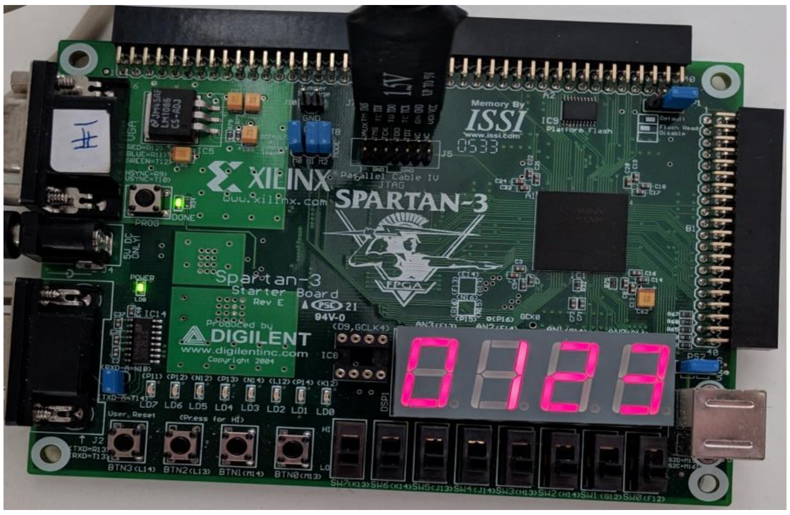
This image shows the multiplexing of the anodes in time and the characters we show in each ascent (constant 0 | 1 | 2 | 3). In addition, we drive the data we want to show at least 2 cycles before driving the anode to have a perfect display of the characters on the LED screen

.

Experiment/Final implementation

During the experiment, the biggest corrections were made to the anti_bounce_reset module, as it was the circuit that had to find both theoretically and practically the appropriate number for the size of the counter so that the button works properly and reacts quickly at our press.

In addition, another change that has been made is the driving of the data in 2 cycles before, instead of being done simultaneously with the ascent. This was because other parts of the '0123' characters were dimly lit on the screen, which means that the LED capacity is too large and this is not allowed..

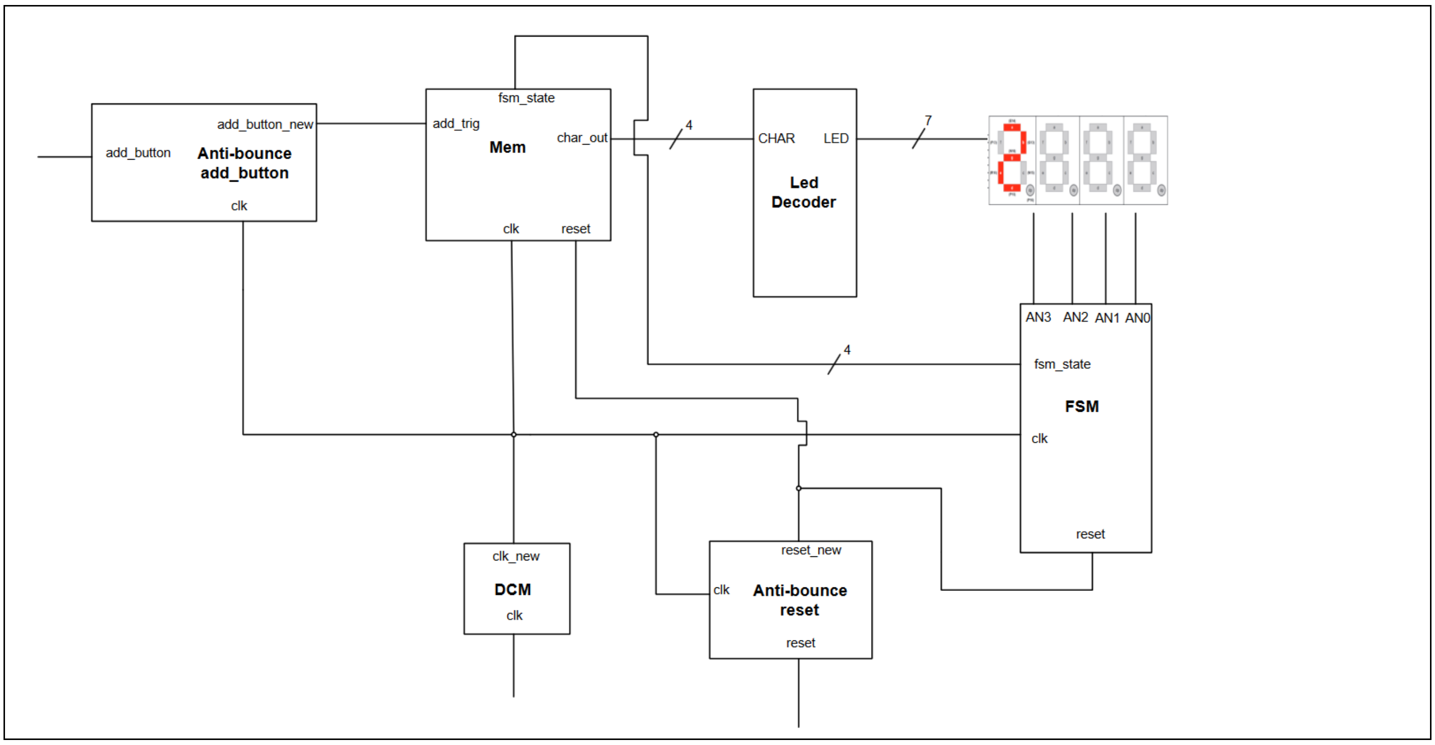


Here is the result from the simulation on the board. The characters are displayed correctly and without another LED being lit..

Part C

Implementation

In this part we extend the design from the previous part and add new modules for the additional functions required.



First we add a module mem that we will have saved the message and it should give us the appropriate characters in the ledDecoder in each clock cycle. To do this we receive a fsm_state signal which is the signal with which we manage the anodes AN3, AN2, AN1, AN0 and light the pieces of the LED screen. This is done 2 cycles before driving the anodes so that the capacities are fully discharged.

Finally in the memory we have an add_trig signal which moves the message one character to the left. This signal, because it originally comes from a button, goes through the anti_bounce module of the previous part which we have made a modification so that the circuit works properly.

Anti-bounce (add_button)

Because the signal add_trig we want to use, we only want to it last to for one cycle we have to make a small modification to the anti_bounce we had.

We add an additional control signal, called pass_through which checks if at the output of the anti_bounce on both flip-flops (sync_1, sync_2) has a logic 1, then we return the out_state to logic 0 and the counter to the value 0.

To do this at the code level means that we will add another if condition where we check each time if the pass_through signal is ΄1΄. In addition, to make sure that by pressing the button continuously we have no issues, we increase the value of COUNTER_DELAY and the size of the counter (the value is 1,048,575 which is equivalent to 0.33secs).

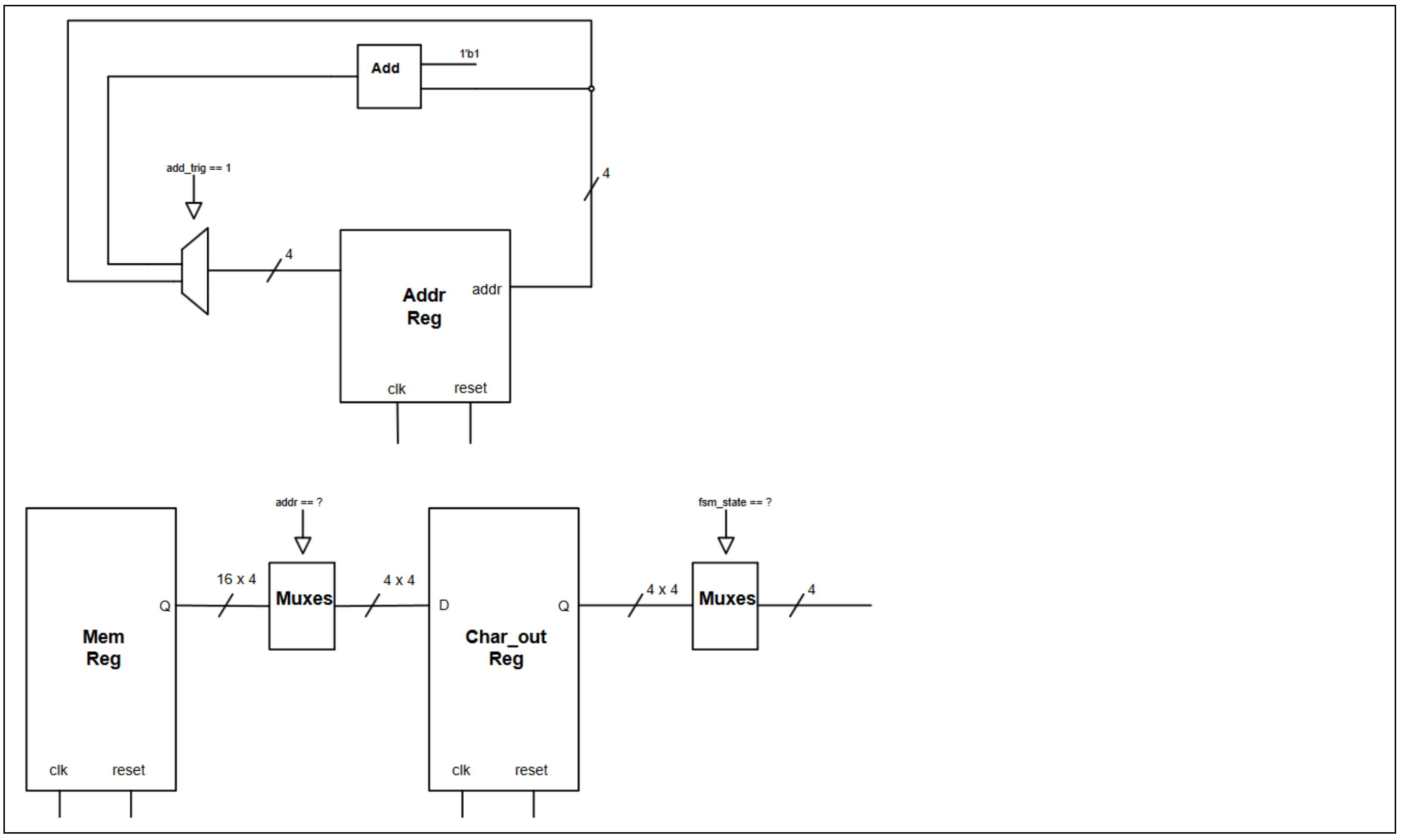
```
module anti_bounce_reset(input clk, in_bit, output reg out_state);
.......
assign pass_through = (sync_1 && sync_2 && out_state);
always @(posedge clk) begin
.......
else if (pass_through ==1'b1) begin  //signal lasts only for a cycle
        out_state<=0;
        counter<=0;
.......
endendmodule
```

Above is the code added to anti_bounce module.

Mem

The mem module is the unit that gives us the characters we want to show, depending on the time we are in and the input from the add_button button.

Below is a functional diagram:



As shown in the figure, we have a counter which we call addr_reg and this is the pointer to the first character we show. If the user presses the button then the value of the counter will automatically increase by one.

Then we have the mem_reg memory that has the characters stored (the message), which after a reset remain constant in each cycle.

These characters go into a combination logic that selects depending on the current address, the next 3 characters that we will show on the screen.

Since we have to light one anode each time, from the 4 characters we show each time we will choose one according to the value of fsm_state which is also the anode control signal.

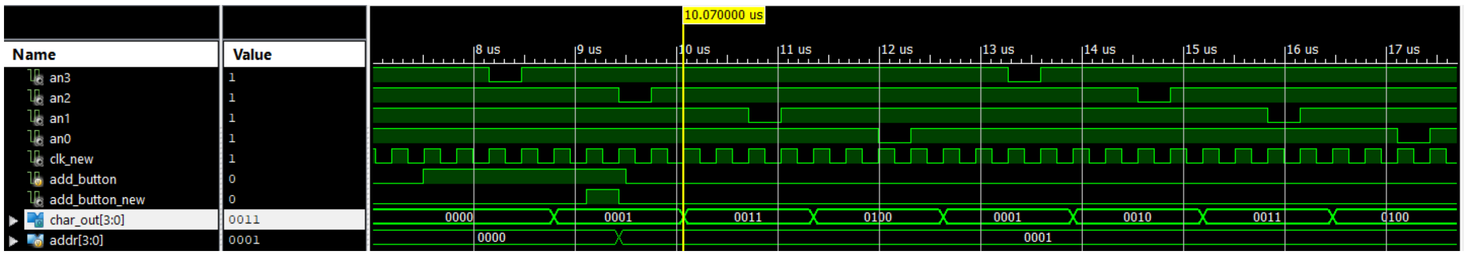
```
module mem(input clock,reset, add_trig,input [3:0]state,output reg [3:0]char_out);
.......
always @(posedge clock or posedge reset) begin
   if(reset == 1'b1)
      addr<=4'b0000;  //initial value
   else if(add_trig)
      addr<=addr+1; //increment
.......
always @(posedge clock or posedge reset) begin
   if(reset == 1'b1) //intialize
      char_out<=4'b0000;
   else if(state == 4'b0000)
      char_out <= message[addr];
   else if(state == 4'b1100)
      char_out <= message[addr+1];
.......
endendmodule
```

Above is the code that implements the mem module. *Message* is the memory of registers where we have stored the message, *addr* the register that we have the pointer in memory and *char_out* the register that will give us the output character based on addr and state.

It is worth mentioning that the schematic given is indicative in terms of behavior and will not be the final circuit after the synthesis of the RTL due to simplifications that will be made by the synthesis tool.

Verification

The testbench from the previous part was used for verification, with the only difference being the addition of an add_button signal for the message scroll button.

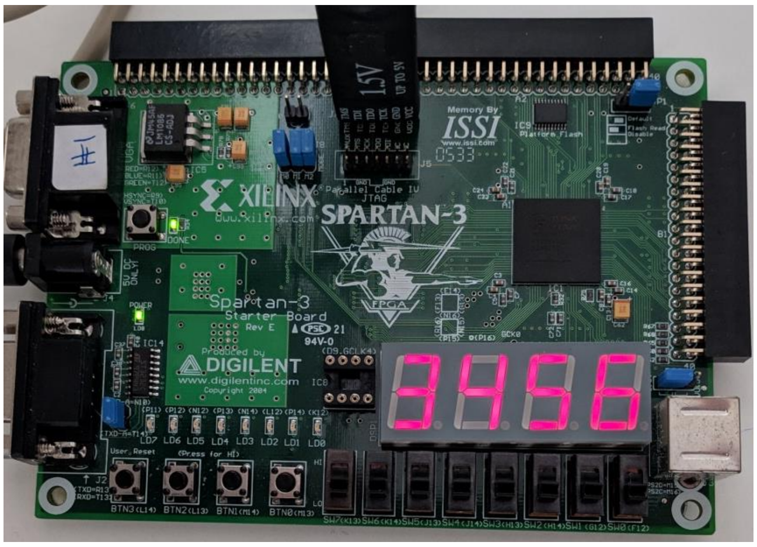


In the simulation image we give an add_button signal which passes to add_button_new and automatically the addr address increases by 1 and essentially the message moves to the left. So from the next row of anodes (yellow cursor) we have the moved message going from 0 | 1 | 2 | 3 to 1 | 2 | 3 | 4 .

Experiment/Final implementation

When testing the board, the only change made is to increase the value of COUNTER_DELAY and the size of the counter.

This way we had the correct operation of the add_button button, ie with one press to have a movement of the message by one character.



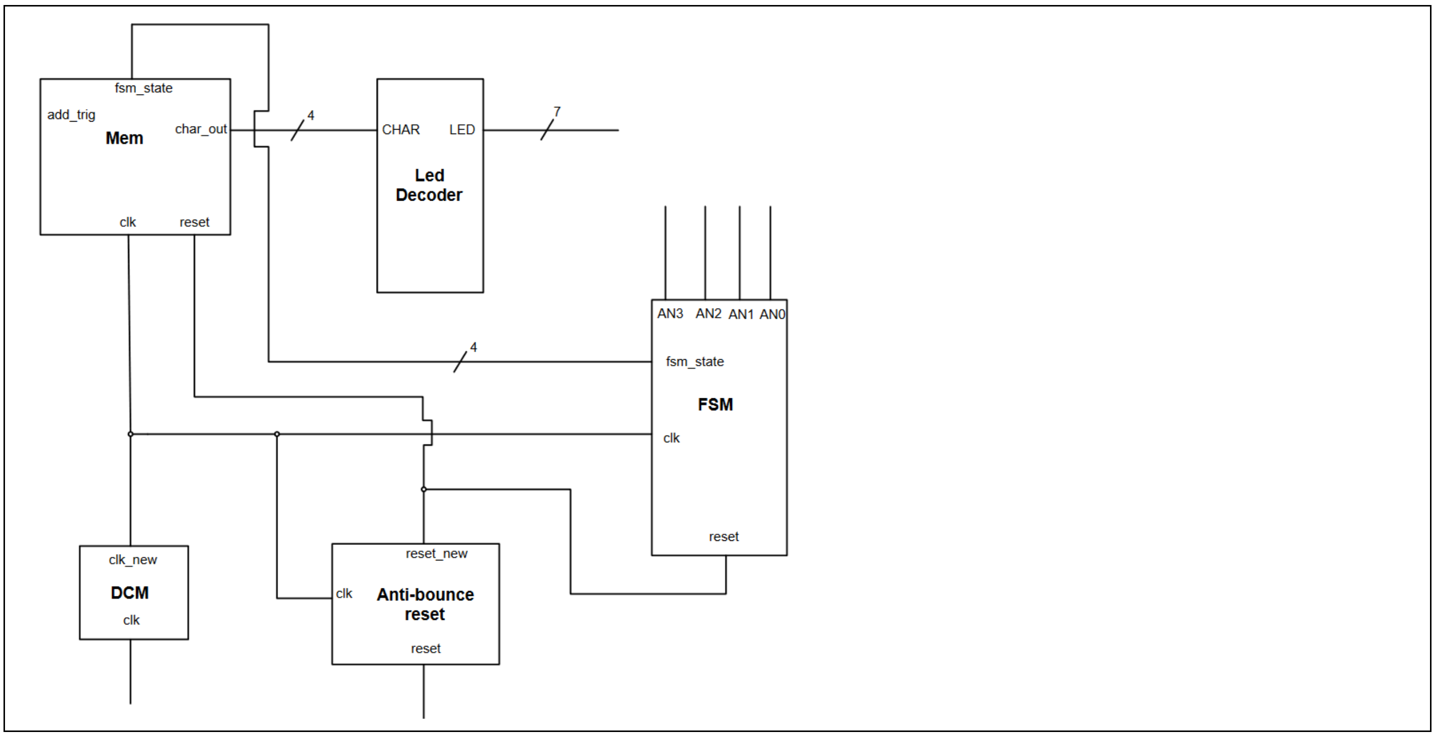
Here is the result of the simulation on the board. The characters that appear are different after a few presses of the add_button button.

Part D

Implementation

In part D we remove the add_button button (hence the anti_bounce-add_button) and add additional functions to the mem module, so that the message is rotated automatically after a certain period of time.

So our final circuit is:



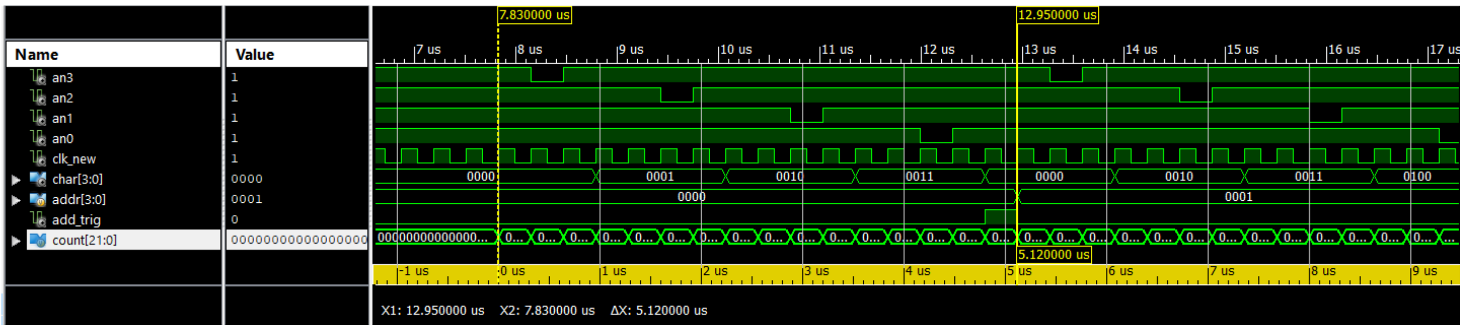
The new mem module, the only difference is that it will now have the add_trig as an internal signal that will activate the clock cycle when the value of a 22-bit counter becomes maximum. This counter is called count.

```
assign add_trig = (count == COUNT_MAX)?1'b1:1'b0;
```

So in the code of the mem module we add the above command and the count that counts up to the parameter COUNT_MAX (ie 2^{22} -1 cycles).

Verification

To verify the correct operation we used the previous testbench from part B.



In the simulation image we see the add_trig signal which is activated when the count reaches the value COUNT_MAX (for the simulation the value is 16). After activating we see the addr increase and the characters displayed on the screen move.

Experiment/Final implementation

In the experimental implementation there was no difficulty since through the simulation it was easy to see that the circuit works properly, so no changes were made to the code after the practical test.

Conclusion

The work was divided into parts and was easier to carry out, however problems arose.

The biggest problems that arose in practice and not in simulation were:

1. Improperly driving the data before driving the anodes, resulting in dull on-screen displays and unclear characters.
2. The appropriate anti-bounce mechanism but also the correct duration of the counter that dealt with the bounces of the button.
3. The initial familiarity with the software tools used for the work .

But after verifications on the board, corrections on the code and our theoretical knowledge all the problems were solved and the laboratory exercise was successful.