Electrical and Computer Engineering
University of Thessaly (UTH)

ECE333 - Digital Systems Lab

Fall Semester — Educational year 2023-2024

Lab01

Seven Segment Display

Georgios Kapakos - AEM: 03165

Contents

Abstract

Using Verilog and the Nexys A7-100T board, we simulate a 4-digit driver. Initially, for **Part A**, we use a decoder to map digit values to 4-bit numbers. Then, for **Part B**, with an MMCM module, we convert the fast input clock into a slow one. To drive the anodes, we use a rotating counter that activates the anodes in one-cold encoding. We initialize our circuit by creating an anti-bounce mechanism to prevent reset bounces. We apply constraints to map the pulses to the board's pins. Additionally, for **Part C**, we create a button that, when pressed, increments each number on the seven-segment display to its next value. Using a memory circuit, we provide the corresponding counter with its next value. We add the BTNR button to our circuit's constraints. Finally, for **Part D**, with a 23-bit counter, we implement a fixed delay of 1.6777214 seconds for changing our digits to their next values.

1 Introduction

The objective of this assignment is to implement a 4-digit 7-segment LED driver on the Nexys A7-100T board, rotating a 16-character message. The message to be rotated consists of the 16 hexadecimal digits, i.e., $0123456789AbCdEF$. The message will be displayed by shifting the characters one by one to the left, appropriately, with:

(1) Pressing a button (BTNR) on the board.

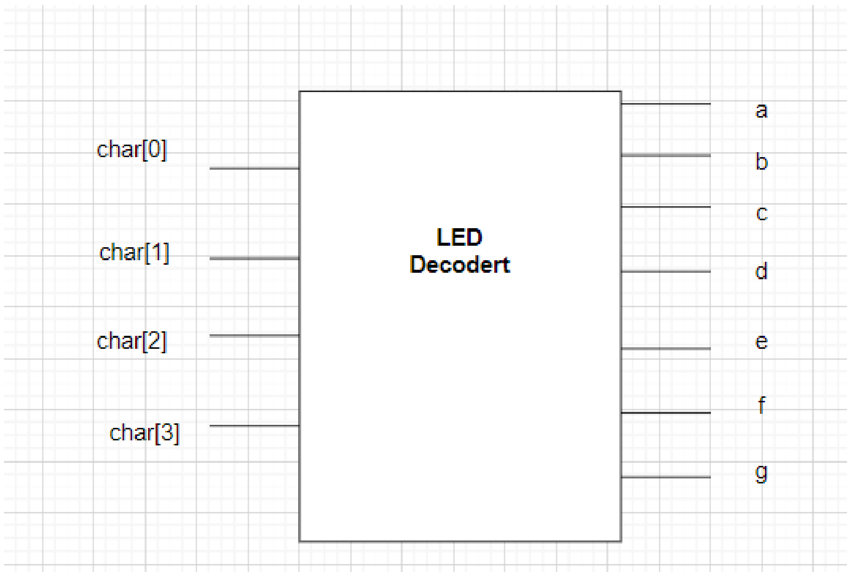
(2) After a specified time interval.

After the last character of the message, the first one follows, effectively rotating the message continuously. We install a fixed delay in the circuit for changing the digits.

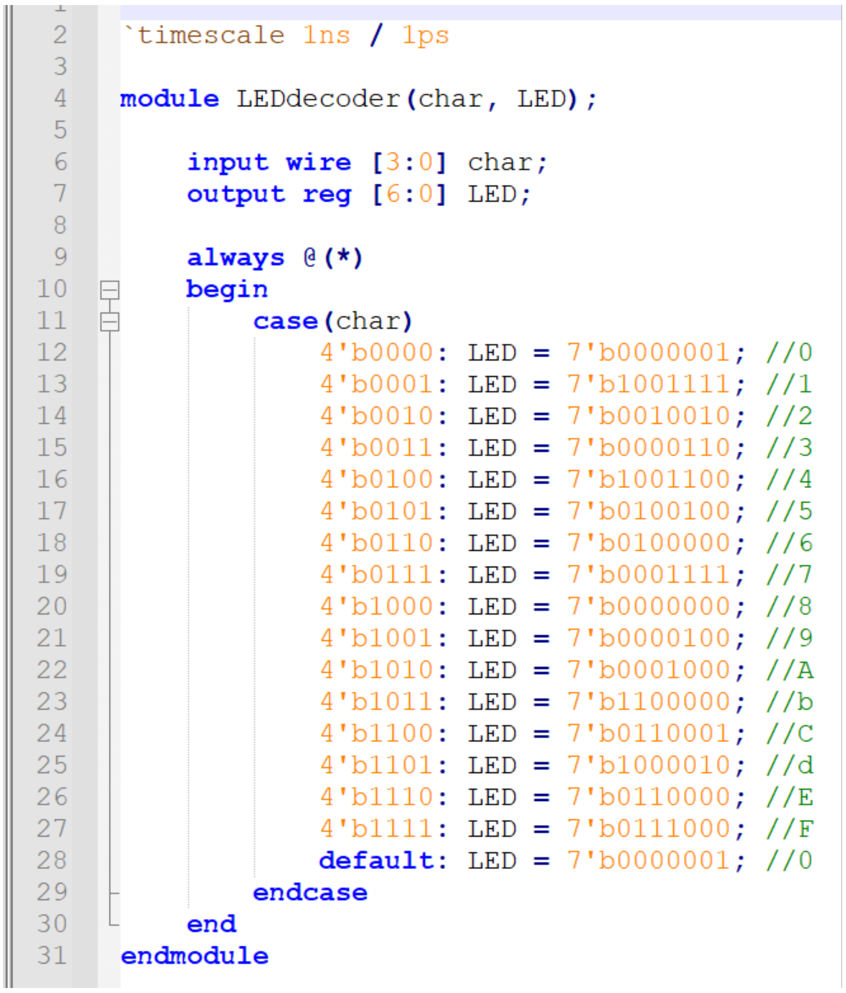
2 Part A - Implementation of the 7-Segment Decoder

2.1 Implementation

We created a decoder with 4 inputs and 7 outputs. It maps each input to a digit from $0123456789abcdef$.



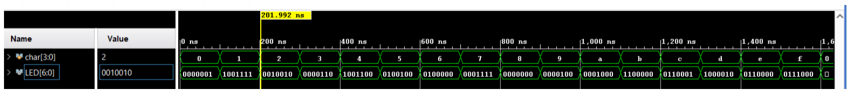
Our circuit consists of an `always` block, which uses a `case` statement to check our input and map it to a specific output.



```verilog
`timescale 1ns / 1ps

module LEDdecoder(char, LED);

    input wire [3:0] char;
    output reg [6:0] LED;

    always @(*)
    begin
        case(char)
            4'b0000: LED = 7'b0000001; //0
            4'b0001: LED = 7'b1001111; //1
            4'b0010: LED = 7'b0010010; //2
            4'b0011: LED = 7'b0000110; //3
            4'b0100: LED = 7'b1001100; //4
            4'b0101: LED = 7'b0100100; //5
            4'b0110: LED = 7'b0100000; //6
            4'b0111: LED = 7'b0001111; //7
            4'b1000: LED = 7'b0000000; //8
            4'b1001: LED = 7'b0000100; //9
            4'b1010: LED = 7'b0001000; //A
            4'b1011: LED = 7'b1100000; //b
            4'b1100: LED = 7'b0110001; //C
            4'b1101: LED = 7'b1000010; //d
            4'b1110: LED = 7'b0110000; //E
            4'b1111: LED = 7'b0111000; //F
            default: LED = 7'b0000001; //0
        endcase
    end
endmodule
```

2.2 Verification

The test framework proved simple. By providing a variety of input values to our decoder, we observed the desired results. We used every distinct input value for the decoder and obtained the outputs as described in the implementation.



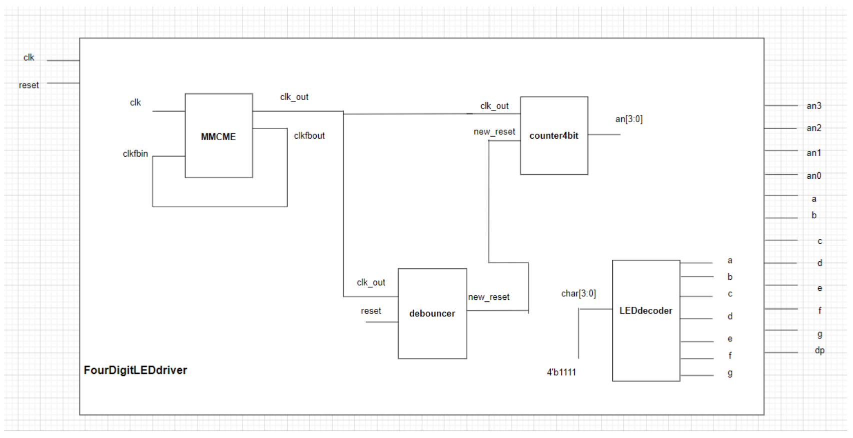
```
Name       Value    0 ns     200 ns            400 ns            600 ns            800 ns            1,000 ns          1,200 ns          1,400 ns          1,6
char[3:0]  2        0        1        2        3        4        5        6        7        8        9        a        b        c        d        e        f        0
LED[6:0]   0010010  0000001  1001111  0010010  0000110  1001100  0100100  0100000  0001111  0000000  0000100  0001000  1100000  0110001  1000010  0110000  0111000
```

2.3 Experiment/Final Implementation

Part A was not implemented on the board.

3 Part B - Driving Four Digits

3.1 Implementation

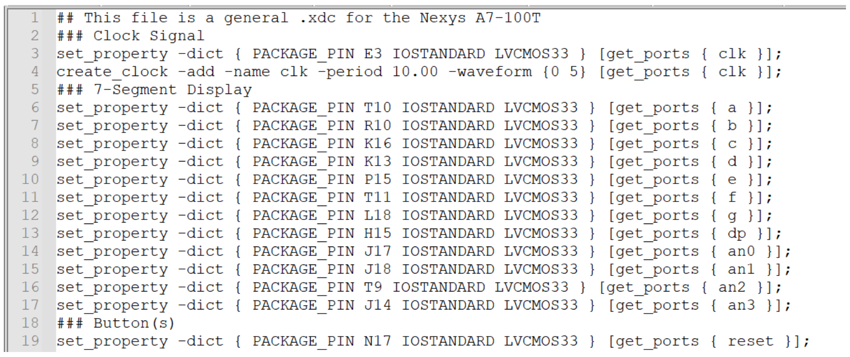


3.1.1 MMCME Module

We implemented an MMCME module that converts the fast clock into a slow one. We set $CLFBOUT\_MULT\_F = 6$, $CLKIN1\_PERIOD = 10$, $CLKOUT1\_DIVIDE = 120$, and $DIVCLK\_DIVIDE = 1$. According to the manual's formula, $F_{out} = F_{clkin} \times M/(D \times O)$, where:

- F_{out}: the new frequency with the slow clock, with a period of 200 ns, thus $F_{out} = 5\,\text{MHz}$,
- F_{clkin}: the frequency of the fast clock, with a period of 10 ns, thus $F_{clkin} = 100\,\text{MHz}$,
- M: corresponds to $CLFBOUT\_MULT\_F$, which determines the multiplication factor for all $CLKOUT$ signals,
- D: corresponds to $DIVCLK\_DIVIDE$, which sets the division factor for all output clocks,
- O: corresponds to $CLKOUT1\_DIVIDE$, which performs the same function as $DIVCLK\_DIVIDE$ but for a specific clock.

$CLKIN1\_PERIOD$ is the period of our input clock, which is 10 ns. We also used a constraints file to connect our Verilog variables to the board's pins.



```
## This file is a general .xdc for the Nexys A7-100T
### Clock Signal
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name clk -period 10.00 -waveform {0 5} [get_ports { clk }];
### 7-Segment Display
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { a }];
set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { b }];
set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { c }];
set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { d }];
set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { e }];
set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { f }];
set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { g }];
set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { dp }];
set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { an0 }];
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { an1 }];
set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { an2 }];
set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { an3 }];
### Button(s)
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports { reset }];
```

3.1.2 MMCME Instantiation

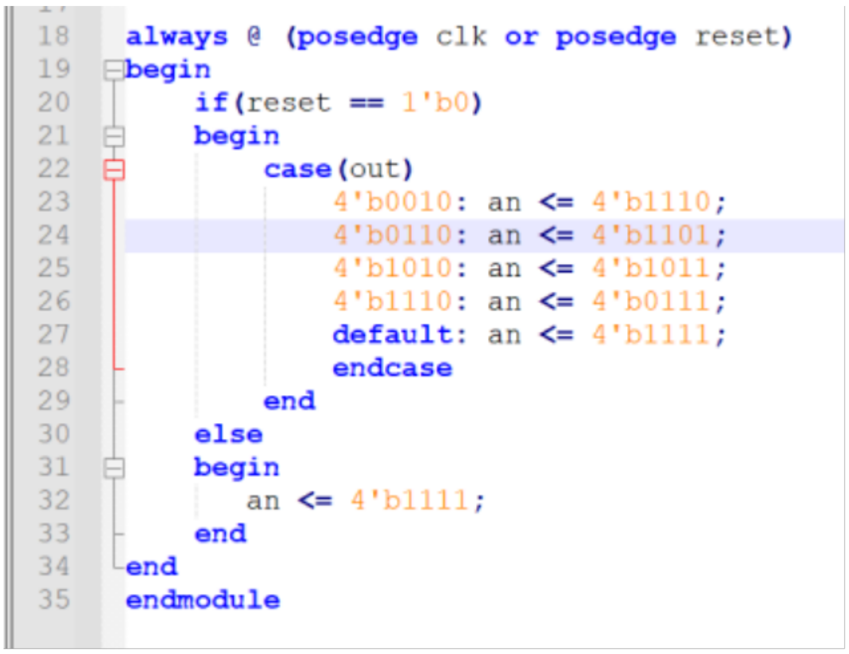
The MMCME instance at the $CLKOUT1$ output takes the value of the new slow clock in our circuit. We short-circuit $CLKFBIN$ with $CLKFBOUT$ because, for external clock alignment, the feedback path clock buffer type must match the forward clock buffer type. It takes the initial fast clock as input at $CLKIN1$ and the reset at the corresponding $RESET$ input.

3.1.3 counter4bit Module

We implemented a 4-bit counter that counts from 0000 to 1111. It activates:

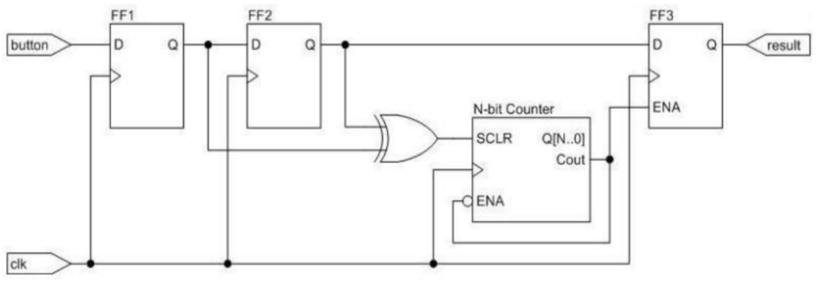
- $an0$ when it reaches 0010,
- $an1$ when it reaches 0110,
- $an2$ when it reaches 1010,
- $an3$ when it reaches 1110,

where $an0, an1, an2, an3$ are the four anodes driving the digits in our circuit.



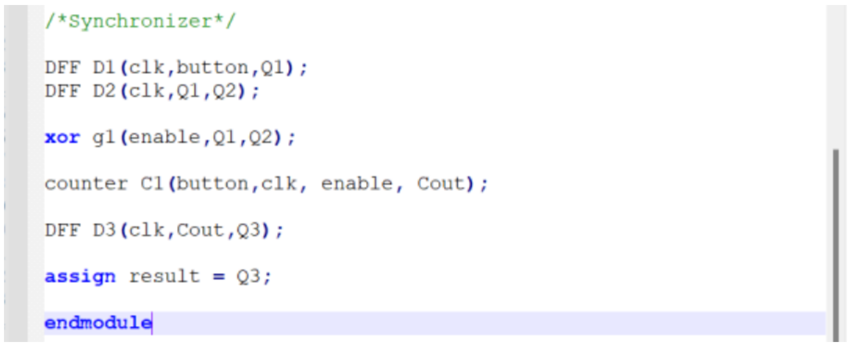
```
always @ (posedge clk or posedge reset)
begin
    if(reset == 1'b0)
    begin
        case(out)
            4'b0010: an <= 4'b1110;
            4'b0110: an <= 4'b1101;
            4'b1010: an <= 4'b1011;
            4'b1110: an <= 4'b0111;
            default: an <= 4'b1111;
            endcase
        end
    else
    begin
        an <= 4'b1111;
    end
end
endmodule
```

3.1.4 debouncer Module



The debouncer is a circuit that synchronizes the signal, counts up to a specific period, and then, considering the signal stable, assigns it the input value. The debouncer consists of:

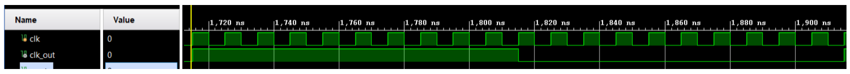
- Two flip-flops and an XOR gate for signal stabilization,
- An 8-bit counter to determine when the signal is considered stable,
- An output flip-flop that passes the signal to the output after further stabilization via the D3 flip-flop.



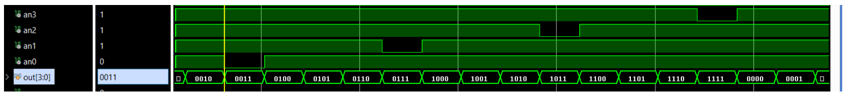
```
/*Synchronizer*/

DFF D1(clk,button,Q1);
DFF D2(clk,Q1,Q2);

xor g1(enable,Q1,Q2);

counter C1(button,clk, enable, Cout);

DFF D3(clk,Cout,Q3);

assign result = Q3;

endmodule
```

3.2 Verification

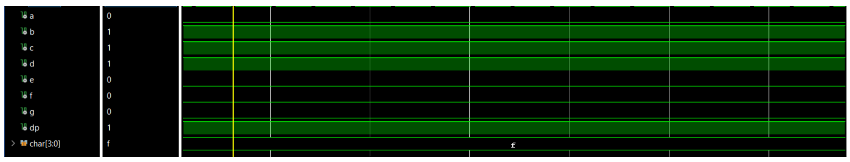
We used a simple testbench that activates the reset once and defines a clock with a 10 ns period. We observed that the new slow clock has a period 20 times larger than the fast clock.



Additionally, $an0$ activates one cycle after the counter reaches 0010, $an1$ after 0110, $an2$ after 1010, and $an3$ after 1110.



Finally, the $FourDigitLEDdriver$ (top-level module) values, with a fixed $char = 1111$, all represent the symbol F across the four digits driven by our anodes.

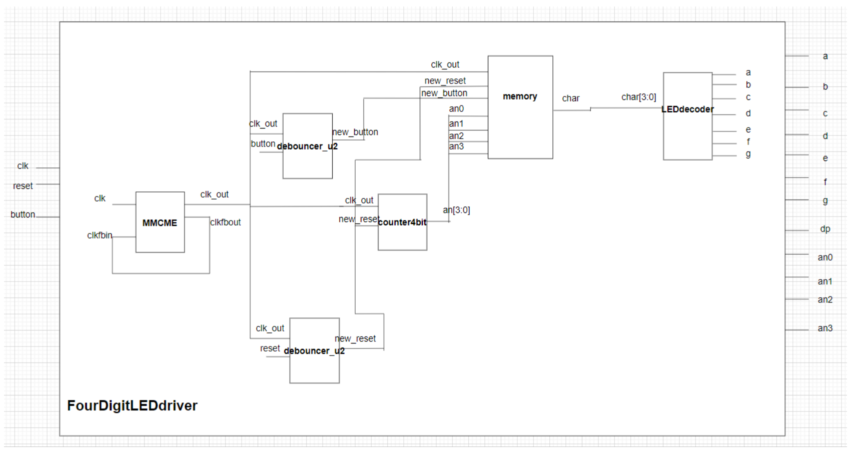


3.3 Experiment/Final Implementation

After implementing the circuit on the board, we observed that all digits of the seven-segment display lit up with the indication $FFFF$, as demonstrated in the testbench. We encountered no issues during implementation on the board.

4 Part C - Stepwise Message Rotation Using a Button

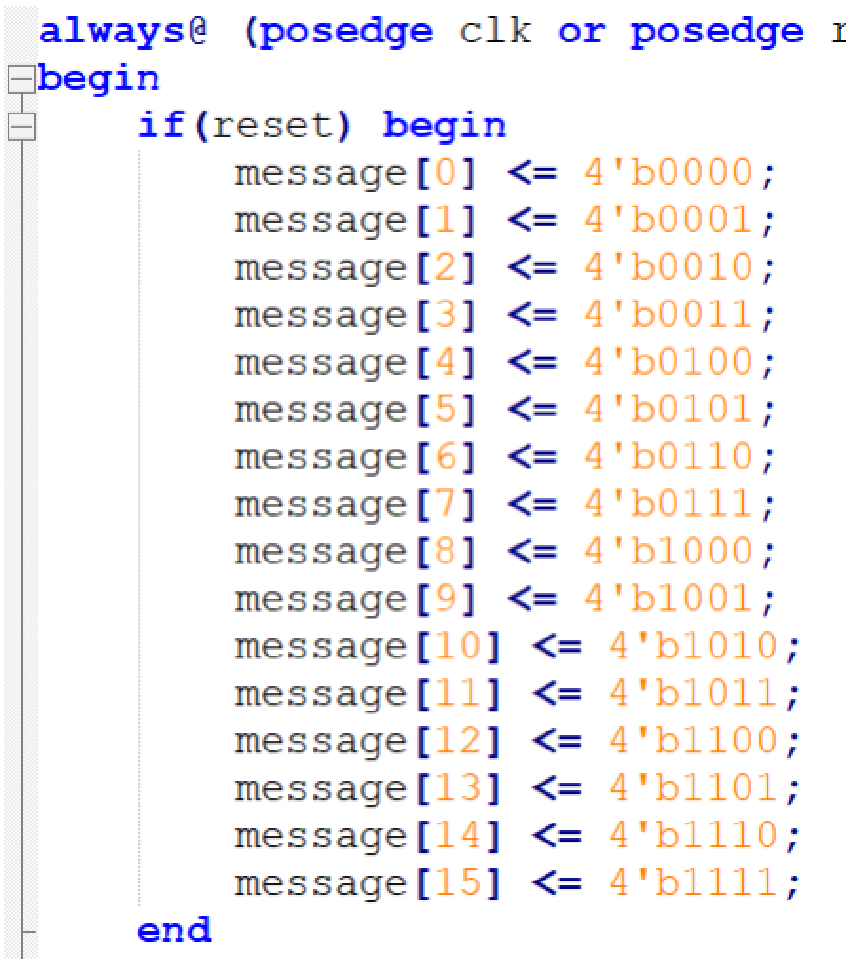
4.1 Implementation



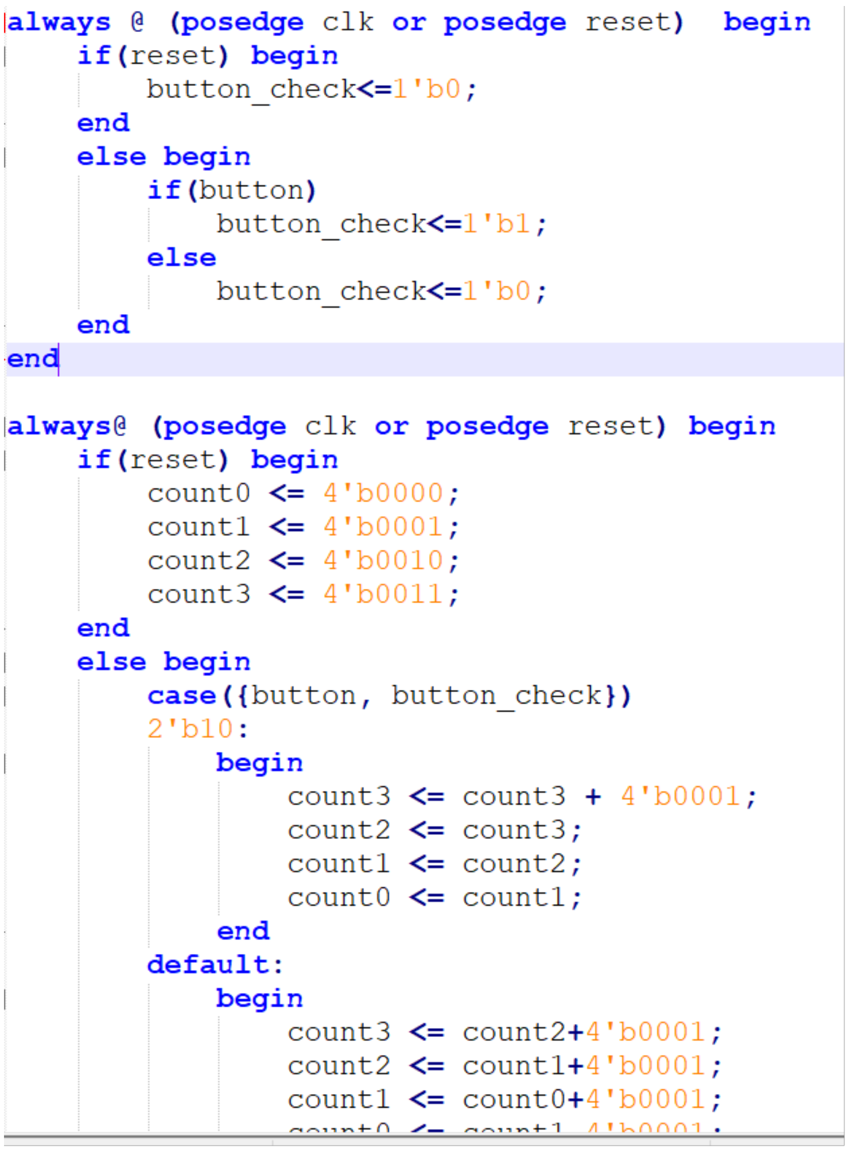
In this part, we created a button that, when pressed, increments our digits to their next value (e.g., from 0123 to 1234). An anti-bounce module was added to the button to avoid bounces during short presses. Since we added the input button to our top-level module, we declared it as a constraint in the constraints file.

```
set_property -dict { PACKAGE_PIN M17 IOSTANDARD LVCMOS33 } [get_ports { button }];
```

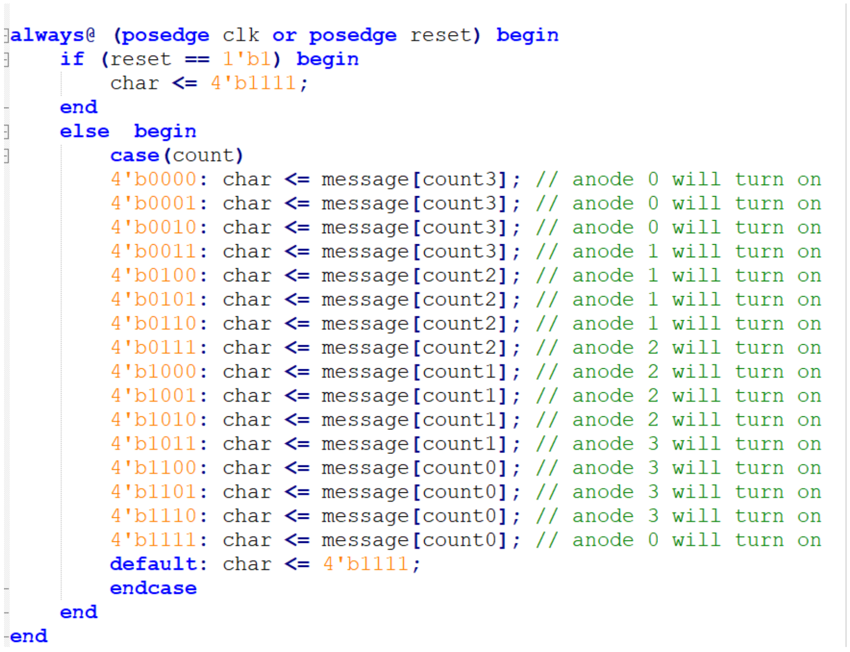
4.1.1 memory Module



```
always@ (posedge clk or posedge r
begin
    if(reset) begin
        message[0] <= 4'b0000;
        message[1] <= 4'b0001;
        message[2] <= 4'b0010;
        message[3] <= 4'b0011;
        message[4] <= 4'b0100;
        message[5] <= 4'b0101;
        message[6] <= 4'b0110;
        message[7] <= 4'b0111;
        message[8] <= 4'b1000;
        message[9] <= 4'b1001;
        message[10] <= 4'b1010;
        message[11] <= 4'b1011;
        message[12] <= 4'b1100;
        message[13] <= 4'b1101;
        message[14] <= 4'b1110;
        message[15] <= 4'b1111;
    end
```



```
always @ (posedge clk or posedge reset)  begin
    if(reset) begin
        button_check<=1'b0;
    end
    else begin
        if(button)
            button_check<=1'b1;
        else
            button_check<=1'b0;
    end
end

always@ (posedge clk or posedge reset) begin
    if(reset) begin
        count0 <= 4'b0000;
        count1 <= 4'b0001;
        count2 <= 4'b0010;
        count3 <= 4'b0011;
    end
    else begin
        case({button, button_check})
        2'b10:
            begin
                count3 <= count3 + 4'b0001;
                count2 <= count3;
                count1 <= count2;
                count0 <= count1;
            end
        default:
            begin
                count3 <= count2+4'b0001;
                count2 <= count1+4'b0001;
                count1 <= count0+4'b0001;
                count0 <= count1 4'b0001;
```

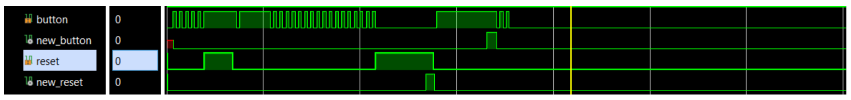


```
always@ (posedge clk or posedge reset) begin
    if (reset == 1'b1) begin
        char <= 4'b1111;
    end
    else  begin
        case(count)
        4'b0000: char <= message[count3]; // anode 0 will turn on
        4'b0001: char <= message[count3]; // anode 0 will turn on
        4'b0010: char <= message[count3]; // anode 0 will turn on
        4'b0011: char <= message[count3]; // anode 1 will turn on
        4'b0100: char <= message[count2]; // anode 1 will turn on
        4'b0101: char <= message[count2]; // anode 1 will turn on
        4'b0110: char <= message[count2]; // anode 1 will turn on
        4'b0111: char <= message[count2]; // anode 2 will turn on
        4'b1000: char <= message[count1]; // anode 2 will turn on
        4'b1001: char <= message[count1]; // anode 2 will turn on
        4'b1010: char <= message[count1]; // anode 2 will turn on
        4'b1011: char <= message[count1]; // anode 3 will turn on
        4'b1100: char <= message[count0]; // anode 3 will turn on
        4'b1101: char <= message[count0]; // anode 3 will turn on
        4'b1110: char <= message[count0]; // anode 3 will turn on
        4'b1111: char <= message[count0]; // anode 0 will turn on
        default: char <= 4'b1111;
        endcase
    end
end
```

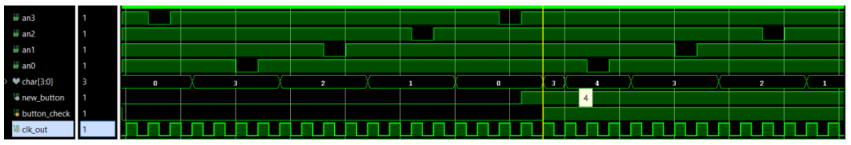
- We have a register array $message$ that takes all possible values from 0000 to 1111, initializing each position with a value.
- To prevent values from changing multiple times while the button is pressed, we created an additional variable $button\_check$, which activates one cycle after pressing the button. We assign the next values to our counters only when $button = 1$ and $button\_check = 0$, ensuring that the values change once per clock period.
- When an anode is active, the output $char$ (the digit displayed on the seven-segment display) takes the value of the message at the address of that anode. When no anode is active, $char$ takes the value F, which is irrelevant as it is never displayed. This is achieved using a `case` statement with a 4-bit counter:
  - $counter$: 0000 $\rightarrow$ 0011: $char$ takes values based on $count3$,
  - $counter$: 0100 $\rightarrow$ 0111: $char$ takes values based on $count2$,
  - $counter$: 1000 $\rightarrow$ 1011: $char$ takes values based on $count1$,
  - $counter$: 1100 $\rightarrow$ 1111: $char$ takes values based on $count0$.

4.2 Verification

The testbench vectors include clk, $reset$, and $button$, with multiple value changes to test the debouncer's functionality based on the duration they remain active. We also cover the changes in the circuit value with button presses, achieving 100% coverage of the button functionality. The debouncer's operation is evident in the reset and button signals, eliminating bounces.



With each button press, the anode values change (e.g. $an0$ from 3, $an1$ from 2, etc.).

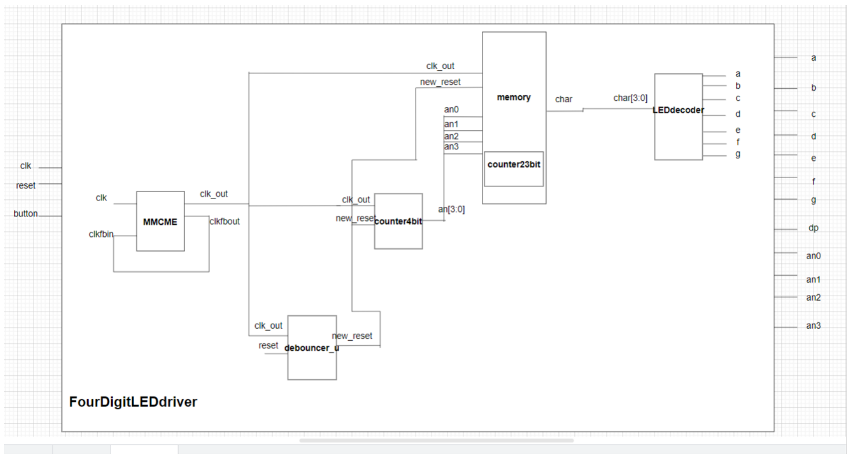


4.3 Experiment/Final Implementation

We tested the program multiple times to identify a functional error due to a misunderstanding of Part C's requirements. No electrical issues arose with the board connection. After correcting the memory module, the board displayed the digits correctly and incremented them with button presses.

5 Part D - Stepwise Message Rotation with Fixed Delay

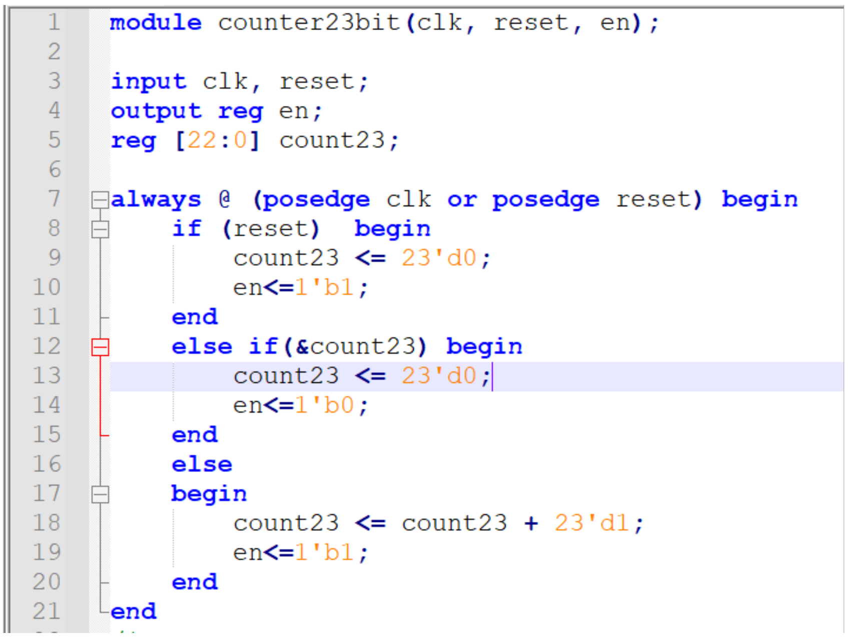
5.1 Implementation



We implemented a circuit where the digits change with a fixed stepwise delay of 1.6777 seconds to their next value (for example, $0123 \to 1234$).

5.1.1 counter23bit Module

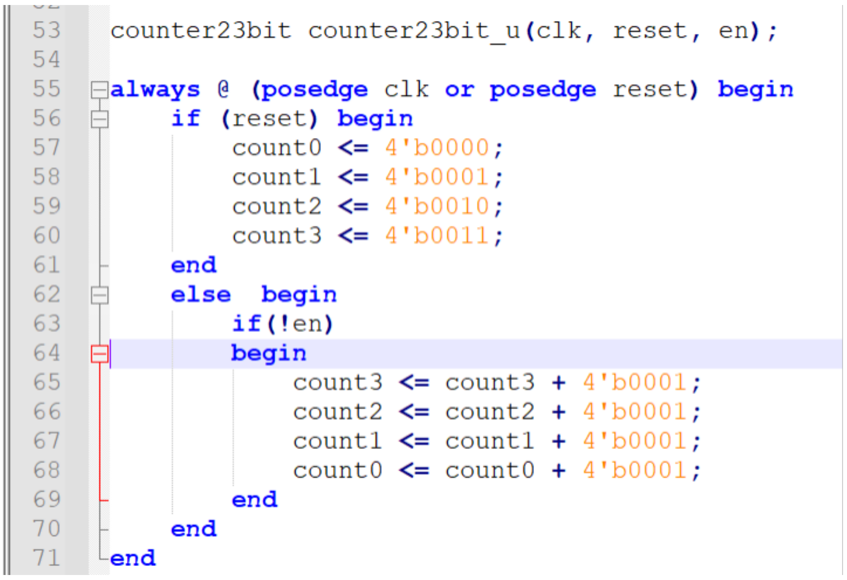
We installed this delay using a 23-bit counter counting from 0 to $2^{23} - 1$. When all bits become 1, we reset it and assert the *en* (enable) signal, shifting the digits to their next state.



```
module counter23bit(clk, reset, en);

input clk, reset;
output reg en;
reg [22:0] count23;

always @ (posedge clk or posedge reset) begin
    if (reset)  begin
        count23 <= 23'd0;
        en<=1'b1;
    end
    else if(&count23) begin
        count23 <= 23'd0;
        en<=1'b0;
    end
    else
    begin
        count23 <= count23 + 23'd1;
        en<=1'b1;
    end
end
```

5.1.2 memory Module

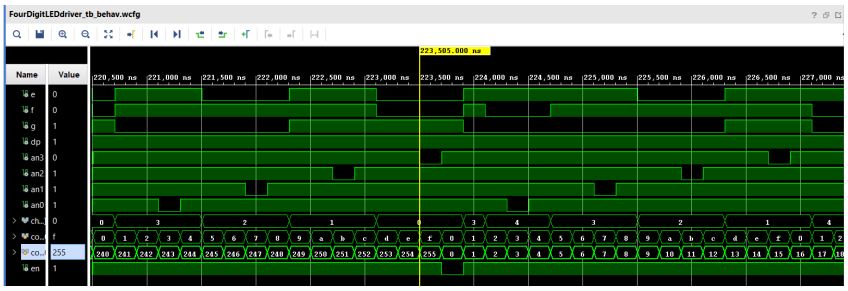
Similar to Part C, except we use the *en* signal from the 23-bit counter to trigger digit changes, activating for one clock cycle.



```verilog
counter23bit counter23bit_u(clk, reset, en);

always @ (posedge clk or posedge reset) begin
    if (reset) begin
        count0 <= 4'b0000;
        count1 <= 4'b0001;
        count2 <= 4'b0010;
        count3 <= 4'b0011;
    end
    else  begin
        if(!en)
        begin
            count3 <= count3 + 4'b0001;
            count2 <= count2 + 4'b0001;
            count1 <= count1 + 4'b0001;
            count0 <= count0 + 4'b0001;
        end
    end
end
```

5.2 Verification

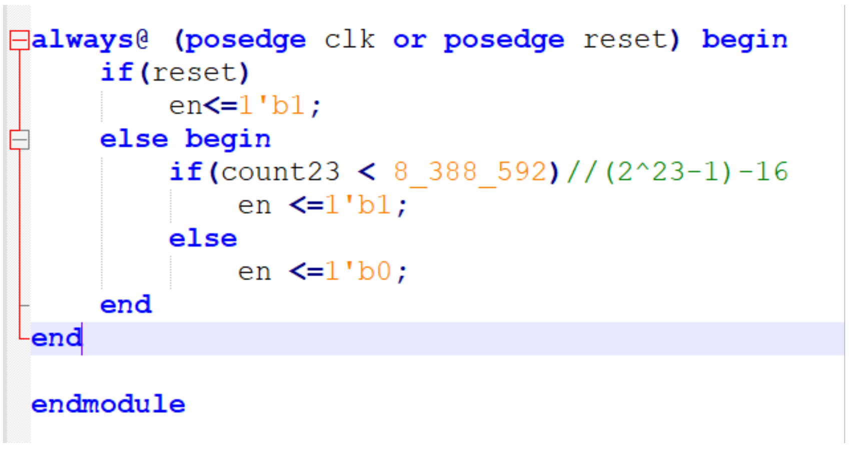
The testbench includes reset changes to further test the debouncer and defines the fast clock at 10 ns. For simulation, we reduced the counter to 8 bits, observing the change of en signal after reaching 255 ($2^8 - 1$), changing the digits accordingly.



We observe that the en signal changes in the next clock cycle, after the last element of our counter, which, locally in the simulation, is 255 ($2^8 - 1$). Following the activation of en, we notice that the digits on the anodes also change value, thus transitioning from 0123 to 1234.

5.3 Experiment/Final Implementation

We attempted to program the FPGA, initially using different code for the 23-bit counter, which was activated if the counter exceeded $(2^{23} - 1) - 16$. On the board, this appeared to never change value and remained fixed at its initial state, i.e., 0123. After modifying the code with the aforementioned `counter23bit`, the program correctly displayed the digit transitions without any issues.



```
always@ (posedge clk or posedge reset) begin
    if(reset)
        en<=1'b1;
    else begin
        if(count23 < 8_388_592)//(2^23-1)-16
            en <=1'b1;
        else
            en <=1'b0;
    end
end

endmodule
```

6 Conclusion

Our laboratory journey began with the design and verification of Parts A and B, which, being straightforward tasks, were handled without significant difficulties. However, Part B presented the challenge of referring to the manual to understand the connections of the MMCME module and its required parameters. This was necessary to determine the appropriate period for the new clock and to create a suitable short circuit between the input buffer clock and the output clock. Beyond these design challenges, the simulation and application on the board had no problems. Subsequently, part C introduced design challenges, which we realized existed only after applying it to the board. Consequently, we revised the code of the `memory` module to ensure the correct operation of the circuit. Finally, Part D presented difficulties in identifying the problem related to the bit changes, which remained static and unchanging. We detected this issue on the board and, after modifying the code, it operated normally without problems.