Lab 5: Stop It

Gabriel Dimas
13 May 2022
Section C

Description

The purpose of this lab is to further improve understanding of State Machines and timing with clock cycles. The goal is to create a game when one set of displays counts up while the other stays constant. The player presses a button when the counting display reaches the constant display value. If the values match, they flash together for four seconds, else they flash alternatively for four seconds. After the flashes, the game resets.

Design

- **Top_Module_Main**
  - Inputs: clkin, btnR, btnU, btnC, btnL
  - Outputs: [6:0]seg, [3:0]an, [15:0]led, dp
  - Implementation: This module adds the others together. This incorporates the **lab5_clks**[1] to slow down the 10MHz clock and apply a Quarter second clock that is high every quarter second with respect to the system clock. These are used as the inputs to the clocks of the other modules. This is the module where the lab checks for equality when the button is pressed. See **Figure 1** the snippet of code of how it is implemented. Use XNOR gates to ensure equality. Also, see **Figure 2** to see how the segments flash. This uses two sets of Flip Flops which alternate or are in sync with each other, depending on the type of flash. The Alternating flip flop alternates between 0 and 1, oppositely. While the synchronous flip flop alternates, but only one is used to assign the anodes to blink synchronously.

```
assign equal[7] = !(CMPUTR[7] ^ USER[7]);
assign equal[6] = !(CMPUTR[6] ^ USER[6]);
assign equal[5] = !(CMPUTR[5] ^ USER[5]);
assign equal[4] = !(CMPUTR[4] ^ USER[4]);
assign equal[3] = !(CMPUTR[3] ^ USER[3]);
assign equal[2] = !(CMPUTR[2] ^ USER[2]);
assign equal[1] = !(CMPUTR[1] ^ USER[1]);
assign equal[0] = !(CMPUTR[0] ^ USER[0]);
assign match = equal[0]&equal[1]&equal[2]&
               equal[3]&equal[4]&equal[5]
               &equal[6]&equal[7];
```

Figure 1 (Left) and Figure 2 (Below)

```
//FLASH ALT
wire alt1, alt2, both1, both2;
FDRE #(.INIT(1'b1) ) ffA1 (.C(clk), .R(1'b0), .CE(qsec&flash_alt), .D(alt1), .Q(alt2));
FDRE #(.INIT(1'b0) ) ffA2 (.C(clk), .R(1'b0), .CE(qsec&flash_alt), .D(alt2), .Q(alt1));

//FLASH_BOTH
FDRE #(.INIT(1'b1) ) ffB1 (.C(clk), .R(1'b0), .CE(qsec&flash_both), .D(both2), .Q(both1));
FDRE #(.INIT(1'b0) ) ffB2 (.C(clk), .R(1'b0), .CE(qsec&flash_both), .D(both1), .Q(both2));
```

[1] Module source from *https://classes.soe.ucsc.edu/cse100/Spring22/lab/lab5/lab5.html*

- **Game_Counter**
  - Inputs: CE, Reset, clk
  - Outputs: [7:0]out
  - Implementation: this module is used for holding a pseudorandom number received from the LFSR. When the game begins, a random number is loaded into the counter, which is then shown on the display for the user to match. This is made with two 4-bit counters. Below is a snippet of the Verilog code of how it is to be implemented.

```
wire [7:0] hold;
countUD4L counter1 (.LD(CE), .Dw(R), .clk(clk), .Q(in[3:0]), .Qout(hold[3:0]));     //left display
countUD4L counter2 (.LD(CE), .Dw(R), .clk(clk), .Q(in[7:4]), .Qout(hold[7:4]));     //right display
```

Figure 3

- **Time_Counter**
  - Inputs: CE, Reset, clk
  - Outputs: [7:0]out
  - Implementation: This counter is implemented as the user's counter. It continuously counts up while in the RunGame is high. It reaches a max value of '3F' and resets to zero. The timer stops when btnU is pressed. This is when equality is established in the **Top_Module_Main** and sent to the State Machine to determine the type of flash. Below is a snippet of the Verilog code for the **TimeCounter**

```
wire start_cnt;
assign start_cnt = UTC & CE & run;
countUD4L counter2 (.Up(start_cnt), .LD(1'b0), .Dw(R), .clk(clk), .Qout(hold[7:4]));
```

Figure 4

- **State_Machine**
  - Inputs: Go, Stop, FourSecs, TwoSecs, Match
  - Outputs: ShowNum, ResetTimer, RunGame, Scored, FlashBoth, FlashAlt
  - Implementation: This is the logic behind the game. This module is made of five flip flops which hold the states of the game. The current state and the previous state are held in memory for the next event to happen. This is also the logic for the outputs of when to show the number, when to reset the timers, when to run the game, when to change the LED, and how to flash the lights when the game ends. See **Figures 7 & 8** for the written format of the state machine. Below is a snippet of the state machine that holds the current and previous state of the game.

```
FDRE #(.INIT(1'b0)) sync1 (.C(clk), .CE(!SEC4), .D(Go), .Q(go));     //supposed to reset when depressed
FDRE #(.INIT(1'b0)) sync2 (.C(clk), .CE(!SEC4), .D(Stop), .Q(stop));         //supposed to reset when depressed

assign Next_IDLE = (IDLE & !go) | (SEC4 & FourSecs);
FDRE #(.INIT(1'b1)) IdleFF (.C(clk), .R(1'b0), .CE(1'b1), .D(Next_IDLE), .Q(IDLE));

assign Next_SEC2 = (SEC2 & !TwoSecs) | (IDLE & go);//| (Delay & !TwoSecs);
FDRE #(.INIT(1'b0)) Sec2FF (.C(clk), .R(1'b0), .CE(1'b1), .D(Next_SEC2), .Q(SEC2));

assign Next_RUN_GAME = (RUN_GAME & !stop) | (SEC2 & TwoSecs);
FDRE #(.INIT(1'b0)) Run_GameFF (.C(clk), .R(1'b0), .CE(1'b1), .D(Next_RUN_GAME), .Q(RUN_GAME));

assign Next_SEC4 = (RUN_GAME & stop) | (SEC4 & !FourSecs);
FDRE #(.INIT(1'b0)) Sec4FF (.C(clk), .R(1'b0), .CE(1'b1), .D(Next_SEC4), .Q(SEC4));
```

Figure 5

- **LFSR**
  - Inputs: clk
  - Outputs: [7:0]num
  - Implementation: This module uses six flip flops and one 3-input XOR gate to produce pseudo-random numbers. This module is constantly running in the background with the clock at the enabler. Although a simple design, this is an important piece for randomizing outputs. When a random number is needed, the **Game_Counter** will just use **LFSR**'s output. See **Appendix** for the logic of the **LFSR**
- **LED_Shifter**
  - Inputs: In, CE, Reset, clk
  - Outputs: [15:0]out
  - Implementation: This module uses a shift register for its logic. When enabled by the state machine, the shift register shifts the queued flip flop to go high. This lights the next LED in the series when a match is received. See below for the Verilog snippet.

```
wire [15:0]led;
FDRE #(.INIT(1'b0) ) ff1  (.C(clk), .R(R), .CE(CE), .D(In),        .Q(led[0]));
FDRE #(.INIT(1'b0) ) ff2  (.C(clk), .R(R), .CE(CE), .D(In&led[0]), .Q(led[1]));
FDRE #(.INIT(1'b0) ) ff3  (.C(clk), .R(R), .CE(CE), .D(In&led[1]), .Q(led[2]));
FDRE #(.INIT(1'b0) ) ff4  (.C(clk), .R(R), .CE(CE), .D(In&led[2]), .Q(led[3]));
FDRE #(.INIT(1'b0) ) ff5  (.C(clk), .R(R), .CE(CE), .D(In&led[3]), .Q(led[4]));
FDRE #(.INIT(1'b0) ) ff6  (.C(clk), .R(R), .CE(CE), .D(In&led[4]), .Q(led[5]));
FDRE #(.INIT(1'b0) ) ff7  (.C(clk), .R(R), .CE(CE), .D(In&led[5]), .Q(led[6]));
FDRE #(.INIT(1'b0) ) ff8  (.C(clk), .R(R), .CE(CE), .D(In&led[6]), .Q(led[7]));
FDRE #(.INIT(1'b0) ) ff9  (.C(clk), .R(R), .CE(CE), .D(In&led[7]), .Q(led[8]));
FDRE #(.INIT(1'b0) ) ff10 (.C(clk), .R(R), .CE(CE), .D(In&led[8]), .Q(led[9]));
FDRE #(.INIT(1'b0) ) ff11 (.C(clk), .R(R), .CE(CE), .D(In&led[9]), .Q(led[10]));
FDRE #(.INIT(1'b0) ) ff12 (.C(clk), .R(R), .CE(CE), .D(In&led[10]), .Q(led[11]));
FDRE #(.INIT(1'b0) ) ff13 (.C(clk), .R(R), .CE(CE), .D(In&led[11]), .Q(led[12]));
FDRE #(.INIT(1'b0) ) ff14 (.C(clk), .R(R), .CE(CE), .D(In&led[12]), .Q(led[13]));
FDRE #(.INIT(1'b0) ) ff15 (.C(clk), .R(R), .CE(CE), .D(In&led[13]), .Q(led[14]));
FDRE #(.INIT(1'b0) ) ff16 (.C(clk), .R(R), .CE(CE), .D(In&led[14]), .Q(led[15]));
```

Figure 6

- **Ring_Counter**
  - Inputs: digsel, clk
  - Outputs: [3:0]out
  - Implementation: See **Design:** hex7Seg *Lab 4: Multiplexers, Full Adders, and Seven Segment Displays*
- **Selector**
  - Inputs: [3:0]in
  - Outputs: [3:0]out
  - Implementation: See **Design:** hex7Seg *Lab 4: Multiplexers, Full Adders, and Seven Segment Displays*
  -
- **hex7seg**
  - Inputs: [3:0]in
  - Outputs: [6:0]out
  - Implementation: See **Design:** hex7Seg *Lab 4: Multiplexers, Full Adders, and Seven Segment Displays*
  -

Testing & Simulation

This lab had its fair share of testing. For starters, making sure that each module was working correctly was extremely important before continuing with another module that relied on the previous one. The **State_Machine** module was one of the more difficult modules to decipher. Ultimately, one was able to finally be created and tested. There were a total of four states: IDLE (the "hold" part of the game in which nothing occurs until a button is pressed), SEC2 (a two-second delay for displaying the pseudorandom number), RUN_GAME (the run of the game after the SEC2 delay, the timer counts up until btnU is pressed), SEC4 (a four-second delay for the lights to flash). The outputs were connected to the states themselves. See **Figure X** below for the implementation of **State_Machine**.

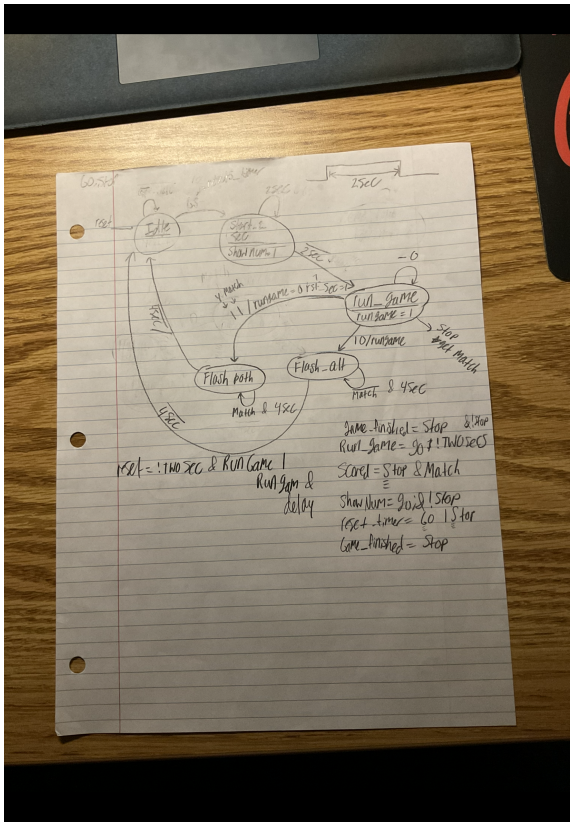


Figure 7

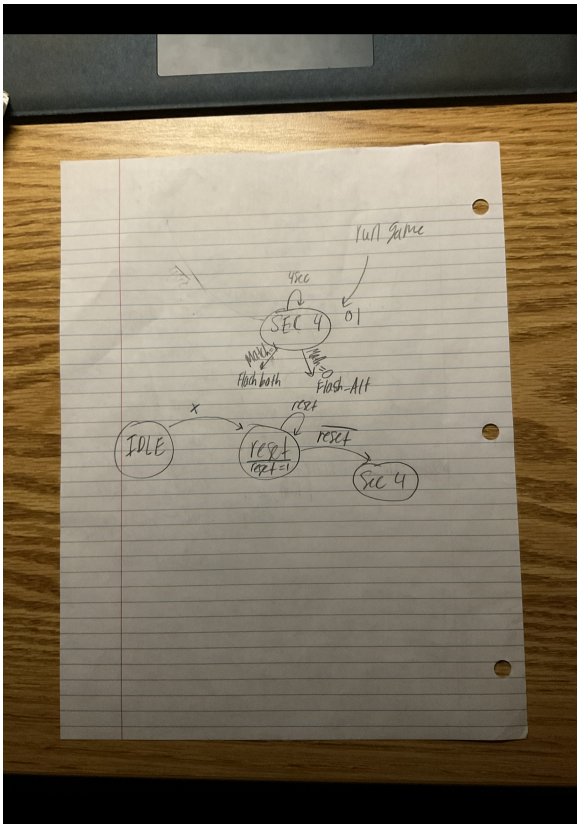


Figure 8

Although the state machine wasn't individually tested, some problems were fixed with a simulation of the machine connected to the other modules. For example, when testing the machine, there was a point where none of the states would go HIGH when the button was pressed. After several hours of trying to figure out the issue, the error came down to the flip flops in the state machine not being clock enabled. The CE was set to 1'b0, indicating that the flip-flops were not enabled. A quick 1'b1 solved that issue and thus concluded the several-hour hunt of the problem.

One of the modules that were created that was seemingly working (and working according to plan) was the timers. The way the timers were implemented the first time was similar to that of a shift register, except that one that was created was inverted with 16 flip flops (to simulate 4 seconds). All but the first flip flops were initialized to '1', and the first was '0'. When the CE is high, the shift register would shift with every cycle of qsec. With CE HIGH, the module would output '1' for 4 seconds until the '0' is reached at the bottom level, then the module would output '0'. This was working correctly for the first cycle of the game but would fail to work for the rest of the time the board was on. This was because, as learned, the shift register would need to be reset after each run of the timer. Also, trying to get around this with the push button and an edge detector connected to the reset of each flip flop would reset the flip flop to INIT 0 values. The final design was to use the **countUD4L**[2] module. This proved to work

[2] Source from *Lab 4: Sequential Logic and Flip-Flops, Gabriel Dimas*

perfectly because if there was a reset input, the flip flop would reset to INIT 0, which is the starting point for a counter. When the final bit is HIGH, the module would output '1', else the module would output '0'. This proved to work and was used in the final design. Ultimately, the testing of the entire lab was a success. After changing the timer to ring counters and adjusting the outputs accordingly, the final results matched those ones in the lab manual.

Results

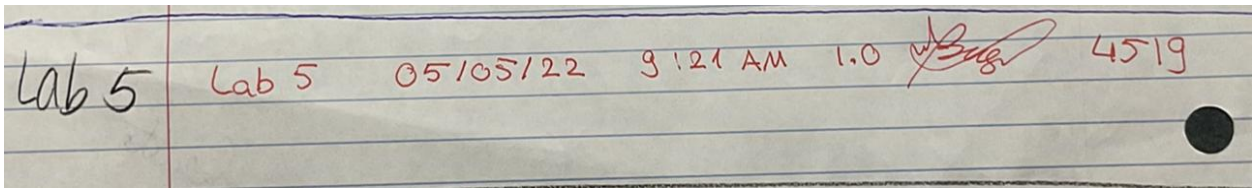
Here are the final states for the state machine

- IDLE: in this state when btnC has not been pressed and is currently in the state. Also in this state when SEC4 time is up and the game is ready to be reset.
- SEC2: State begins when btnC is pressed and initiates two seconds. In this state as long as the two seconds have not elapsed.
- RUN_GAME: State begins when TwoSecs is LOW. The game counter begins to count up. In this state until btnU is pressed.
- SEC4: State begins when in RUN_GAME and btnU is pressed. Also in the state, while 4 seconds have not elapsed. When four seconds elapsed, it goes LOW, and the next state is IDLE.

Conclusion

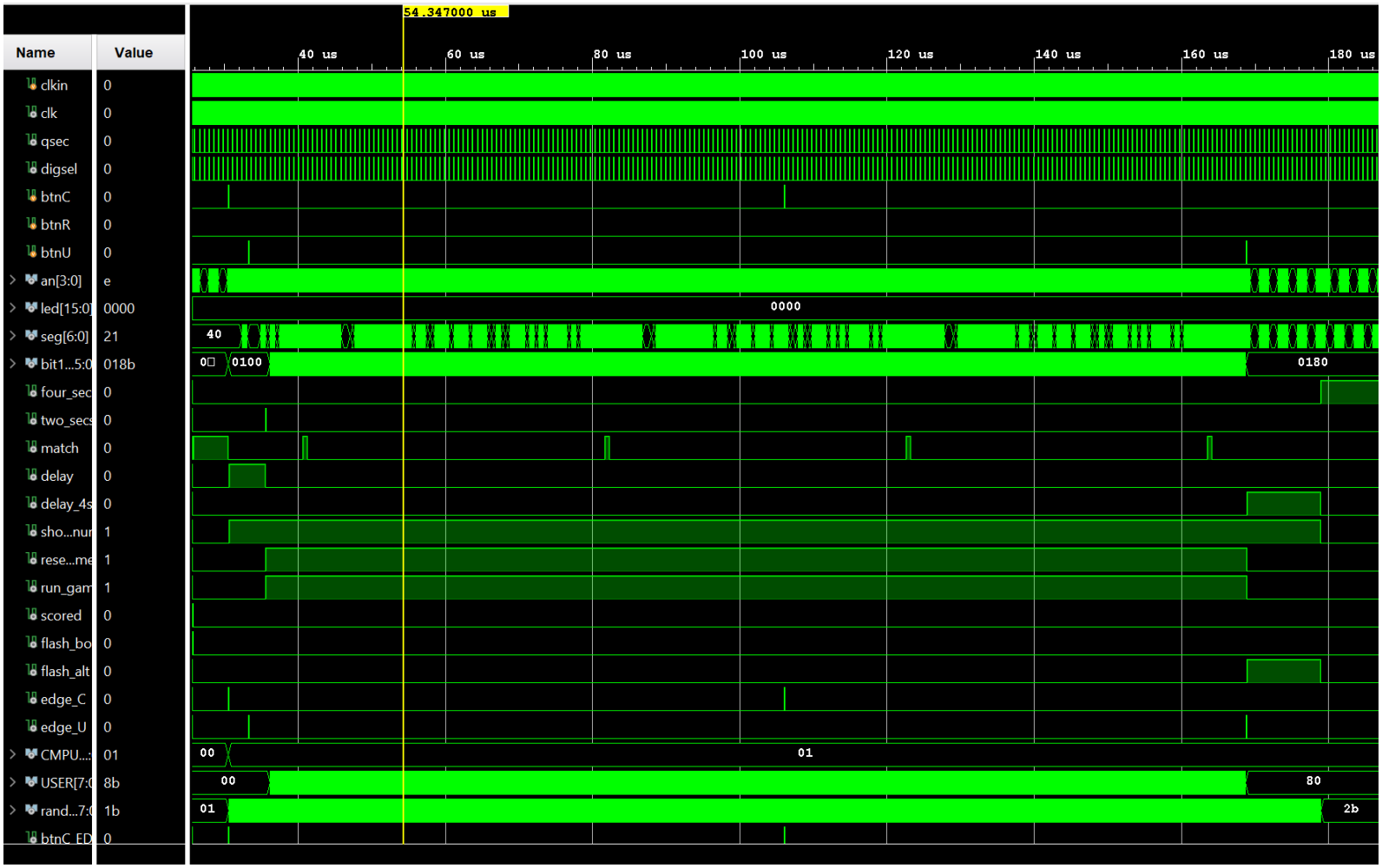
This lab is very relevant for real-world applications. Most of the older generation games only work on one set of memory that doesn't have the ability to be expanded. For example, a game that is only meant for one task would only need a set amount of memory. Because of this, only a specific amount would be needed and only certain digital logic would be needed to implement. State machines are the logic behind many games. With a well-defined state, one would be able to build very large projects with one brain driving the entire project. The important part about making state machines is knowing how they work and to ensure that each possibility of the machine is covered. If done again, it should be clear to know that every state (including states that may or be used) should be included when building. This proved extremely helpful when deciding which states would be necessary or which would just be equivalent to others.

Appendix



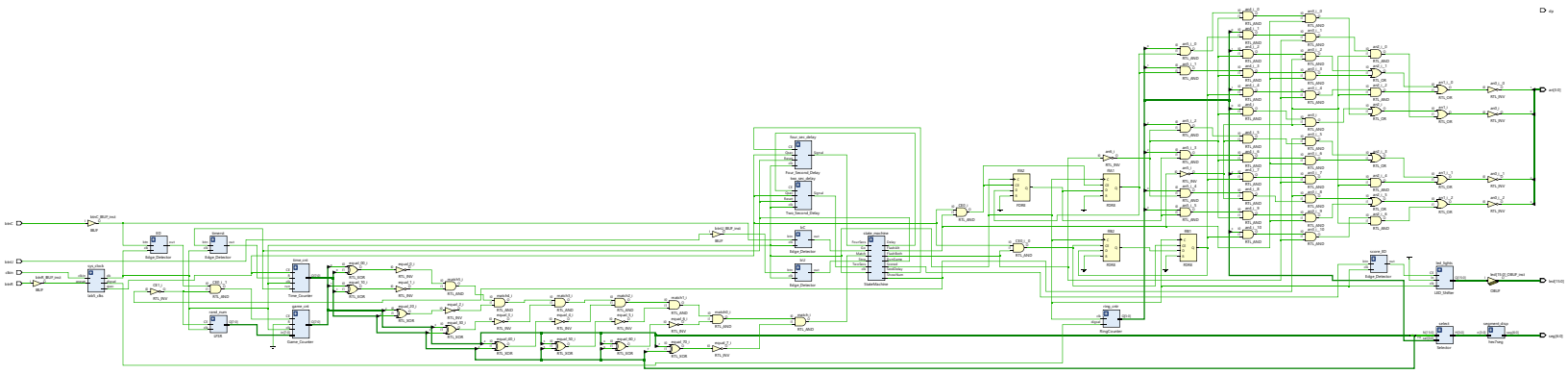
05/05/2022 LAB CODE: 4519

Below are the Lab 5 Schematics, Verilog Code, and Waveform of simulation results.

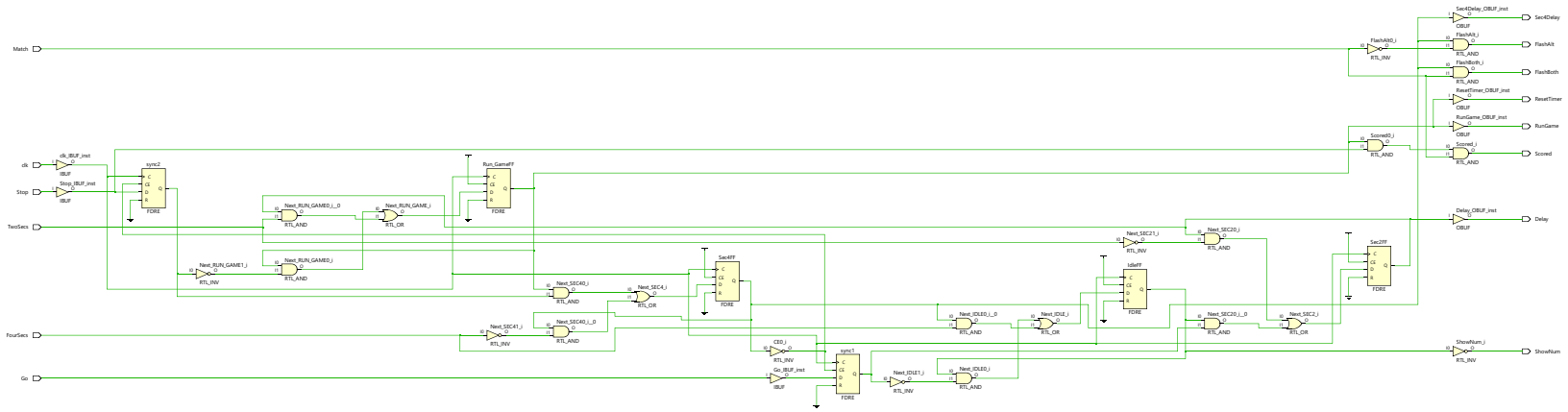


Waveform of the Top Level Module during simulation

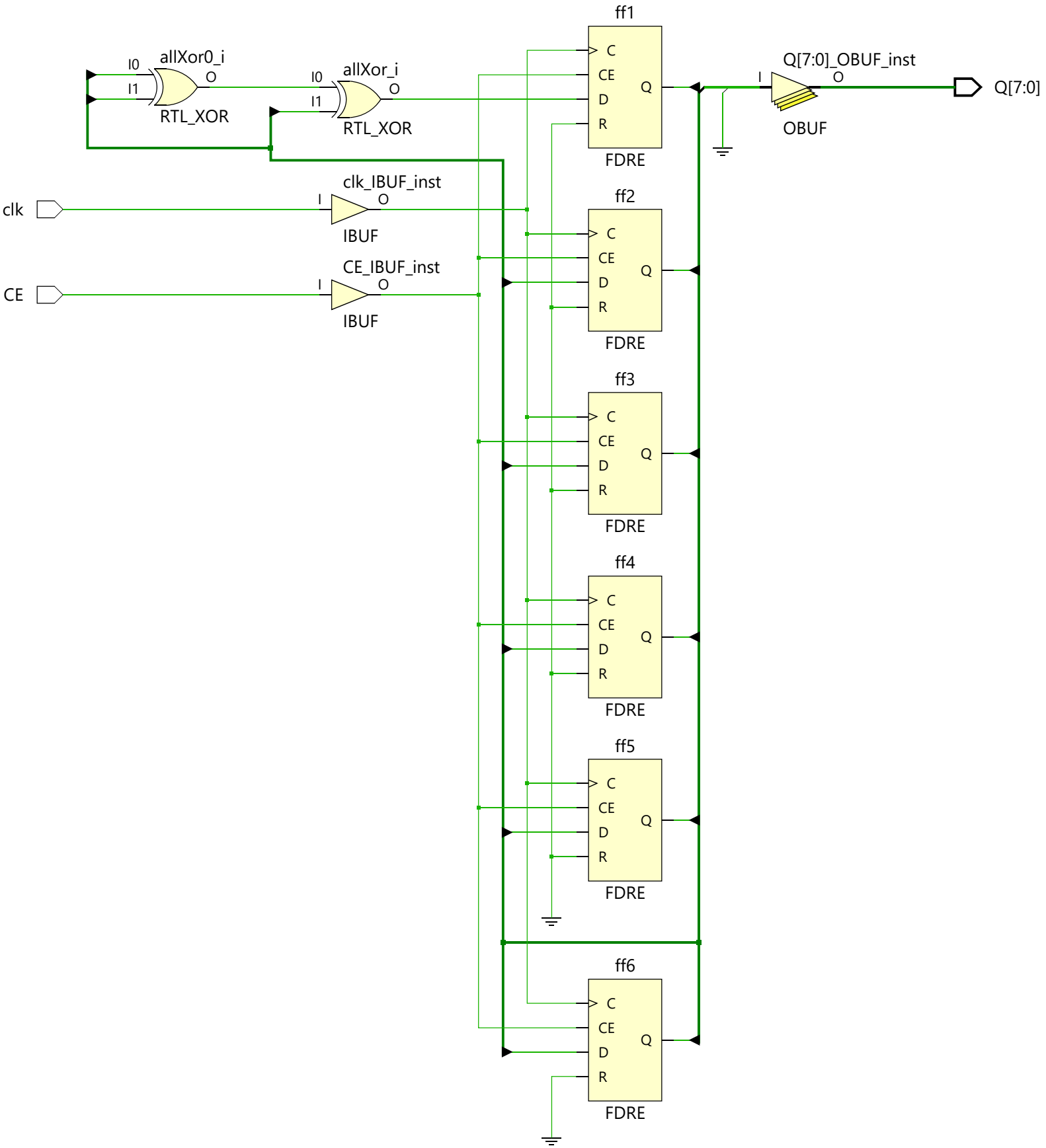
Top_Module_Main



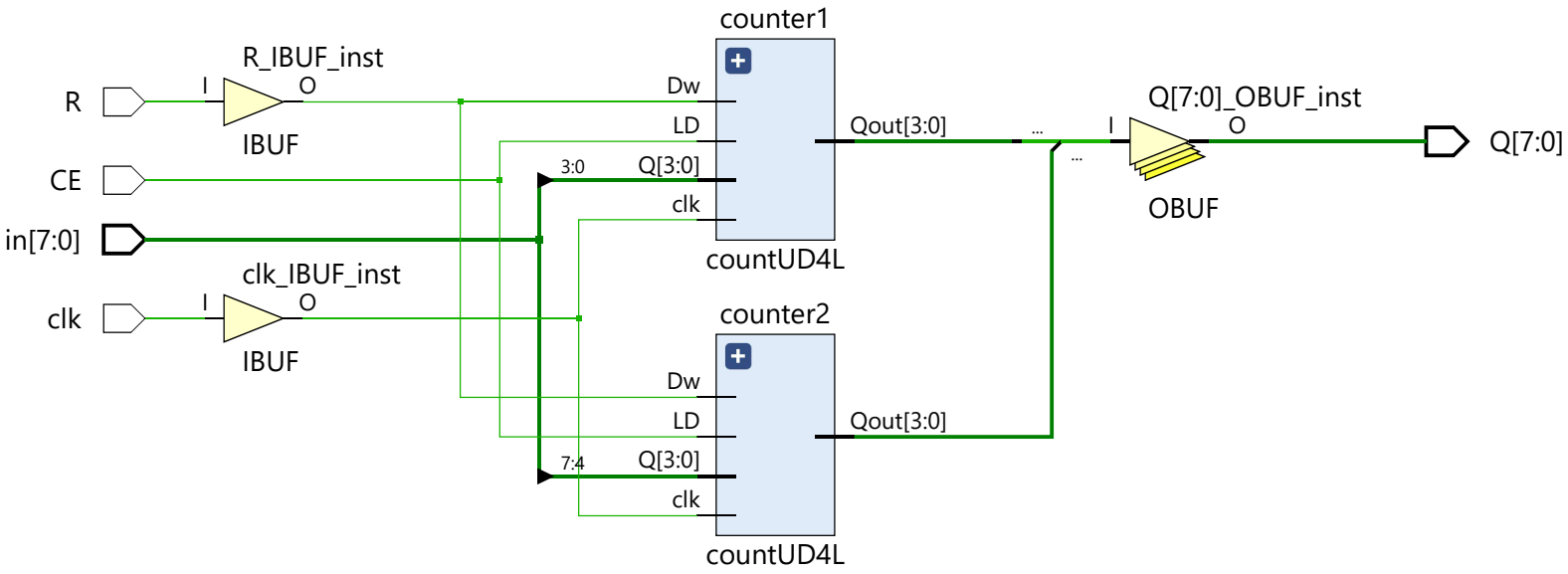
State_Machine



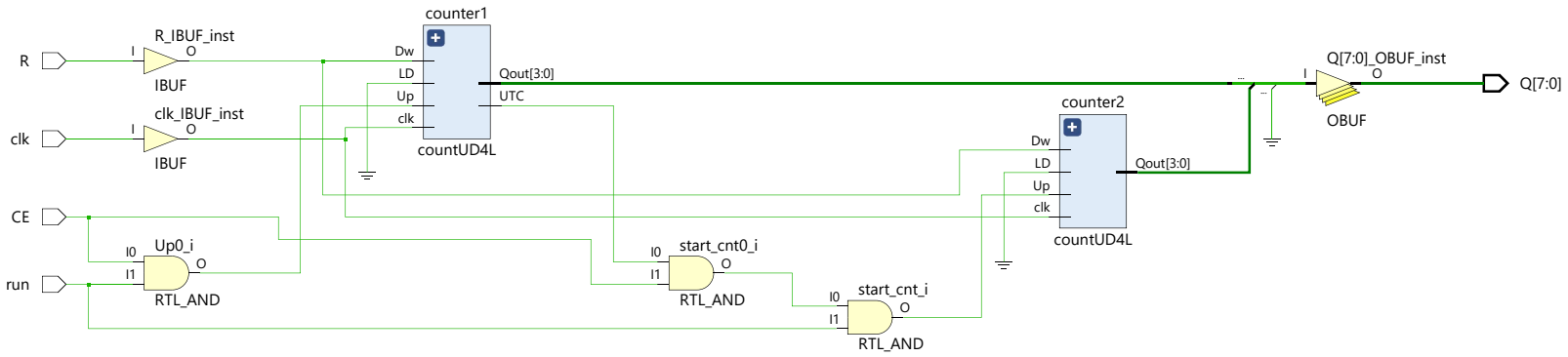
LFSR


ff1
C
CE
D
R
Q
FDRE
allXor0_i
I0
I1
O
RTL_XOR
allXor_i
I0
I1
O
RTL_XOR
Q[7:0]_OBUF_inst
I
O
OBUF
Q[7:0]
clk
clk_IBUF_inst
I
O
IBUF
CE
CE_IBUF_inst
I
O
IBUF
ff2
ff3
ff4
ff5
ff6

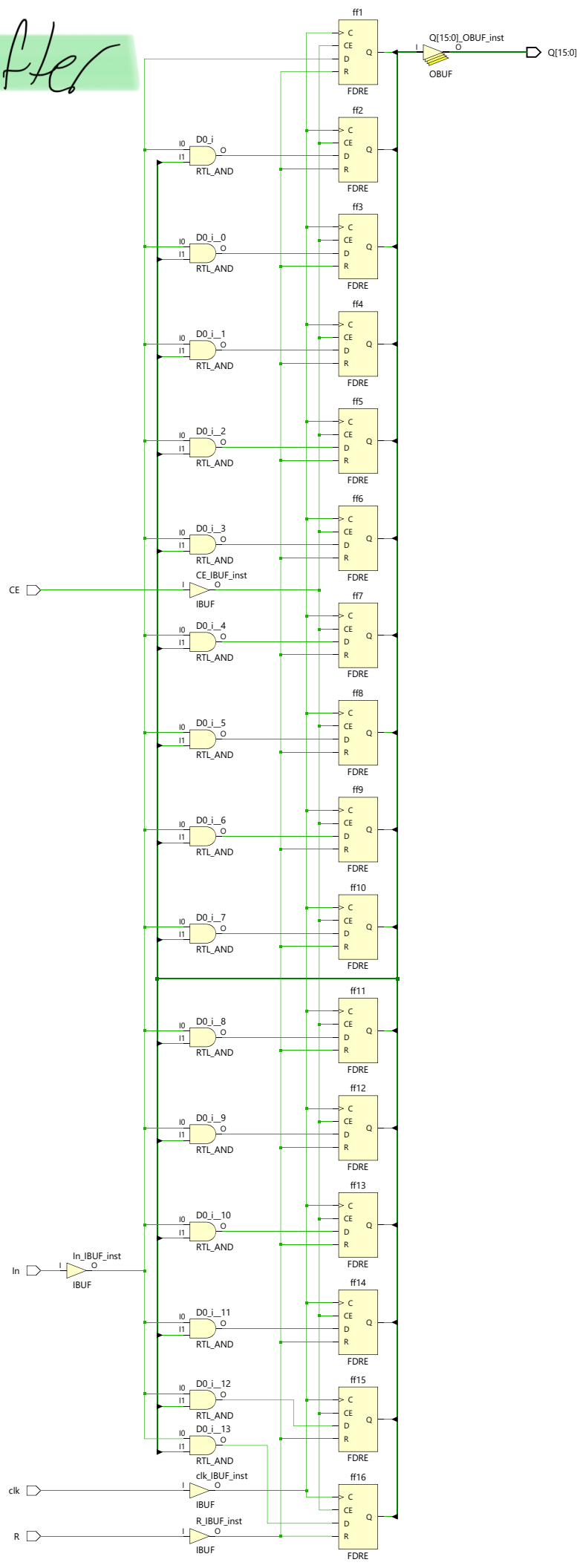
Game_Counter



Time_Counter



LED_Shifter


ff1
FDRE
Q[15:0]_OBUF_inst
OBUF
Q[15:0]
ff2
D0_i
RTL_AND
FDRE
ff3
D0_i_0
RTL_AND
FDRE
ff4
D0_i_1
RTL_AND
FDRE
ff5
D0_i_2
RTL_AND
FDRE
ff6
D0_i_3
RTL_AND
FDRE
CE
CE_IBUF_inst
IBUF
ff7
D0_i_4
RTL_AND
FDRE
ff8
D0_i_5
RTL_AND
FDRE
ff9
D0_i_6
RTL_AND
FDRE
ff10
D0_i_7
RTL_AND
FDRE
ff11
D0_i_8
RTL_AND
FDRE
ff12
D0_i_9
RTL_AND
FDRE
ff13
D0_i_10
RTL_AND
FDRE
In
In_IBUF_inst
IBUF
ff14
D0_i_11
RTL_AND
FDRE
ff15
D0_i_12
RTL_AND
D0_i_13
RTL_AND
FDRE
clk
clk_IBUF_inst
IBUF
ff16
R
R_IBUF_inst
IBUF
FDRE

```
module Top_Module_Main(
    input clkin,
    input btnR,
    input btnU,
    input btnC,
    output [15:0] led,
    output [3:0] an,
    output dp,
    output [6:0] seg
    );
    wire clk, digsel, qsec;
    wire [15:0] bit16out;
    lab5_clks sys_clock
(.clkin(clkin), .greset(btnR),
.clk(clk), .digsel(digsel),
.qsec(qsec));

    wire four_secs, two_secs, match,
delay, delay_4sec;

    wire show_num, reset_timer,
```

```
run_game, scored, flash_both,
flash_alt;
    wire edge_C, edge_U;
    Edge_Detector bC (.clk(clk),
.btn(btnC), .out(edge_C));
    Edge_Detector bU (.clk(clk),
.btn(btnU), .out(edge_U));
    StateMachine state_machine
(.Go(edge_C), .Stop(edge_U),
.FourSecs(four_secs),
.TwoSecs(two_secs), .Match(match),
.ShowNum(show_num),

.ResetTimer(reset_timer),
.RunGame(run_game), .Scored(scored),
.Delay(delay),

.Sec4Delay(delay_4sec),
.FlashBoth(flash_both),
.FlashAlt(flash_alt), .clk(clk));
```

```
    Two_Second_Delay  two_sec_delay
(.clk(clk), .CE(delay),
.Reset(run_game),  .Qsec(qsec),
.Signal(two_secs));
    Four_Second_Delay four_sec_delay
(.clk(clk), .CE(delay_4sec),
.Reset(run_game),  .Qsec(qsec),
.Signal(four_secs));


    wire [7:0] CMPUTR, USER;
    wire [7:0] rand_ints;
    LFSR rand_num (.clk(clk),
.CE(!show_num), .Q(rand_ints));

   wire btnC_ED;
   Edge_Detector ED (.clk(clk),
.btn(btnC), .out(btnC_ED));
   Game_Counter game_cnt (.CE(btnC_ED
& !show_num), .R(1'b0),
.in(rand_ints), .clk(clk),
```

```
.Q(CMPUTR));   //from LFSR changed to
reset_timer

    wire [7:0]equal;
    wire rstTimer;
    Edge_Detector timerst (.clk(clk),
.btn(run_game), .out(rstTimer));
    Time_Counter time_cnt (.CE(qsec),
.R(rstTimer), .run(run_game),
.clk(clk), .Q(USER)); //added a RESET
    assign equal[7] = !(CMPUTR[7] ^
USER[7]);
    assign equal[6] = !(CMPUTR[6] ^
USER[6]);
    assign equal[5] = !(CMPUTR[5] ^
USER[5]);
    assign equal[4] = !(CMPUTR[4] ^
USER[4]);
    assign equal[3] = !(CMPUTR[3] ^
USER[3]);
    assign equal[2] = !(CMPUTR[2] ^
```

```
USER[2]);
    assign equal[1] = !(CMPUTR[1] ^
USER[1]);
    assign equal[0] = !(CMPUTR[0] ^
USER[0]);
    assign match =
equal[0]&equal[1]&equal[2]&

equal[3]&equal[4]&equal[5]
                    &equal[6]&equal[7];

    assign bit16out[15:8] = CMPUTR;
    assign bit16out[7:0] = USER;


    wire [15:0]led_shift;
    wire scored_ED;
    Edge_Detector score_ED
(.clk(clk), .btn(scored),
.out(scored_ED));
    LED_Shifter led_lights
```

```
(.In(1'b1), .CE(scored_ED),
.clk(clk), .Q(led_shift));

    //FLASH ALT
    wire alt1, alt2, both1, both2;
    FDRE #(.INIT(1'b1) ) ffA1
(.C(clk), .R(1'b0),
.CE(qsec&flash_alt), .D(alt1),
.Q(alt2));
    FDRE #(.INIT(1'b0) ) ffA2
(.C(clk), .R(1'b0),
.CE(qsec&flash_alt), .D(alt2),
.Q(alt1));

    //FLASH_BOTH
    FDRE #(.INIT(1'b1) ) ffB1
(.C(clk), .R(1'b0),
.CE(qsec&flash_both), .D(both2),
.Q(both1));
    FDRE #(.INIT(1'b0) ) ffB2
(.C(clk), .R(1'b0),
```

```
.CE(qsec&flash_both), .D(both1),
.Q(both2));

    //below is from lab 4
    wire [3:0]Qring;
    RingCounter ring_cntr
(.digsel(digsel), .clk(clk),
.Q(Qring));

    assign led = led_shift;
    assign an[3] =
!(show_num&(Qring[3] & !flash_both &
!flash_alt) | (Qring[3] &alt1 &
!flash_both & flash_alt) | (Qring[3]
&both1 & flash_both & !flash_alt));
    assign an[2] =
!(show_num&(Qring[2] & !flash_both &
!flash_alt) | (Qring[2] &alt1 &
!flash_both & flash_alt) | (Qring[2]
&both1 & flash_both & !flash_alt));
    assign an[1] = !((Qring[1] &
```

```
!flash_both & !flash_alt) | (Qring[1]
&alt2 & !flash_both & flash_alt) |
(Qring[1] &both1 & flash_both &
!flash_alt));
    assign an[0] = !((Qring[0] &
!flash_both & !flash_alt) | (Qring[0]
&alt2 & !flash_both & flash_alt) |
(Qring[0] &both1 & flash_both &
!flash_alt));

    wire [3:0]sel;
    Selector select(.sel(Qring),
.N(bit16out), .H(sel));
    hex7seg segment_disp (.n(sel),
.seg(seg));

endmodule
```

```
//status: APPEARS TO WORK
module LED_Shifter(
    input In,
    input CE,
    input R,
    input clk,
    output [15:0] Q
    );

    wire [15:0]led;
    FDRE #(.INIT(1'b0) ) ff1
(.C(clk), .R(R), .CE(CE), .D(In),
   .Q(led[0]));
    FDRE #(.INIT(1'b0) ) ff2
(.C(clk), .R(R), .CE(CE),
.D(In&led[0]), .Q(led[1]));
    FDRE #(.INIT(1'b0) ) ff3
(.C(clk), .R(R), .CE(CE),
.D(In&led[1]), .Q(led[2]));
    FDRE #(.INIT(1'b0) ) ff4
(.C(clk), .R(R), .CE(CE),
```

```
.D(In&led[2]), .Q(led[3]));
    FDRE #(.INIT(1'b0) ) ff5
(.C(clk), .R(R), .CE(CE),
.D(In&led[3]), .Q(led[4]));
    FDRE #(.INIT(1'b0) ) ff6
(.C(clk), .R(R), .CE(CE),
.D(In&led[4]), .Q(led[5]));
    FDRE #(.INIT(1'b0) ) ff7
(.C(clk), .R(R), .CE(CE),
.D(In&led[5]), .Q(led[6]));
    FDRE #(.INIT(1'b0) ) ff8
(.C(clk), .R(R), .CE(CE),
.D(In&led[6]), .Q(led[7]));
    FDRE #(.INIT(1'b0) ) ff9
(.C(clk), .R(R), .CE(CE),
.D(In&led[7]), .Q(led[8]));
    FDRE #(.INIT(1'b0) ) ff10
(.C(clk), .R(R), .CE(CE),
.D(In&led[8]), .Q(led[9]));
    FDRE #(.INIT(1'b0) ) ff11
(.C(clk), .R(R), .CE(CE),
```

```
.D(In&led[9]), .Q(led[10]));
    FDRE #(.INIT(1'b0) ) ff12
(.C(clk), .R(R), .CE(CE),
.D(In&led[10]), .Q(led[11]));
    FDRE #(.INIT(1'b0) ) ff13
(.C(clk), .R(R), .CE(CE),
.D(In&led[11]), .Q(led[12]));
    FDRE #(.INIT(1'b0) ) ff14
(.C(clk), .R(R), .CE(CE),
.D(In&led[12]), .Q(led[13]));
    FDRE #(.INIT(1'b0) ) ff15
(.C(clk), .R(R), .CE(CE),
.D(In&led[13]), .Q(led[14]));
    FDRE #(.INIT(1'b0) ) ff16
(.C(clk), .R(R), .CE(CE),
.D(In&led[14]), .Q(led[15]));
    assign Q = led;
endmodule
```

```
//status: IN PROGRESS
module Time_Counter(
    input CE,
    input R,
    input run,
    input clk,
    output [7:0] Q
    );
    wire UTC; wire [7:0] hold;

    countUD4L counter1 (.Up(CE&run),
.Dw(R), .LD(1'b0), .clk(clk),
.UTC(UTC), .Qout(hold[3:0]));

    wire start_cnt;
    assign start_cnt = UTC & CE & run;
    countUD4L counter2
(.Up(start_cnt), .LD(1'b0), .Dw(R),
.clk(clk), .Qout(hold[7:4]));
   // assign Q = hold;
   assign Q[7] = hold[4], Q[6] =
```

```
hold[5], Q[5] = 1'b0, Q[4] = 1'b0,
Q[3] = hold[0], Q[2] = hold[1], Q[1]
= hold[2], Q[0] = hold[3];
endmodule
```

```
//status: IN PROGRESS
module Game_Counter(
    input CE,
    input R,
    input clk,
    input [7:0] in,
    output [7:0] Q
    );
    //Sw is the reset
    wire [7:0] hold;
    countUD4L counter1 (.LD(CE),
.Dw(R), .clk(clk), .Q(in[3:0]),
.Qout(hold[3:0]));    //left display
    countUD4L counter2 (.LD(CE),
.Dw(R), .clk(clk), .Q(in[7:4]),
.Qout(hold[7:4]));    //right display
    assign Q[0] = hold[3];
    assign Q[1] = hold[2];
    assign Q[2] = hold[1];
    assign Q[3] = hold[0];
```

```
    assign Q[7] = hold[7];
    assign Q[6] = hold[6];
    assign Q[5] = hold[5];
    assign Q[4] = hold[4];




endmodule
```

```
//status: IN PROGRESS
module StateMachine(
    input Go,
    input Stop,
    input FourSecs,     //always
counting four_secs
    input TwoSecs,      //always
counting two_secs
    input Match,        //HIGH if
show_num module equals time_counter,
else LOW
    input clk,
    output ShowNum,
    output ResetTimer,
    output RunGame,
    output Delay,
    output Sec4Delay,
    output Scored,
    output FlashBoth,
    output FlashAlt
    );
```

```
    wire IDLE, SEC2, RUN_GAME,
SEC4;//, FLASH_BOTH, FLASH_ALT;
    wire Next_IDLE, Next_SEC2,
Next_RUN_GAME, Next_SEC4;//,
Next_FLASH_BOTH, Next_FLASH_ALT;
    wire go, stop;
    FDRE #(.INIT(1'b0)) sync1
(.C(clk), .CE(!SEC4), .D(Go),
.Q(go));     //supposed to reset when
depressed
    FDRE #(.INIT(1'b0)) sync2
(.C(clk), .CE(!SEC4), .D(Stop),
.Q(stop));          //supposed to
reset when depressed

    assign Next_IDLE = (IDLE & !go) |
(SEC4 & FourSecs);
    FDRE #(.INIT(1'b1)) IdleFF
(.C(clk), .R(1'b0), .CE(1'b1),
.D(Next_IDLE), .Q(IDLE));
```

```
    assign Next_SEC2 = (SEC2 &
!TwoSecs) | (IDLE & go);//| (Delay &
!TwoSecs);
    FDRE #(.INIT(1'b0)) Sec2FF
(.C(clk), .R(1'b0), .CE(1'b1),
.D(Next_SEC2), .Q(SEC2));

    assign Next_RUN_GAME = (RUN_GAME
& !stop) | (SEC2 & TwoSecs);
    FDRE #(.INIT(1'b0)) Run_GameFF
(.C(clk), .R(1'b0), .CE(1'b1),
.D(Next_RUN_GAME), .Q(RUN_GAME));

    assign Next_SEC4 = (RUN_GAME &
stop) | (SEC4 & !FourSecs);
    FDRE #(.INIT(1'b0)) Sec4FF
(.C(clk), .R(1'b0), .CE(1'b1),
.D(Next_SEC4), .Q(SEC4));

    assign ResetTimer = RUN_GAME;// |
(IDLE & !FourSecs & !Delay)| (IDLE &
```

```
!TwoSecs & !Delay);//| (FLASH_BOTH &
!FourSecs) | (FLASH_ALT & !FourSecs);
//big Go and Stop so it only runs one
time
    assign ShowNum = !IDLE;
    assign Delay =  SEC2; //SEC2 |
SEC4;
    assign Sec4Delay = SEC4;//SEC2 |
SEC4;//SEC4 | (TwoSecs&!Delay);
    assign RunGame = RUN_GAME;
    assign Scored = RUN_GAME & Stop &
Match;   //Stop is the button and
Match is the correct match
    assign FlashAlt = SEC4 & !Match;
    assign FlashBoth = SEC4 & Match;



endmodule
```