University of California Santa Cruz
Department of Computer Engineering
Lab Experiment Report # 5

Timing Game

Author: Kyle Jeffrey
Lab Partner: NA
Due Date: 2/21/18

Objective

Implement a single player game running a 6-bit clock that the player tries to stop on the given value. The lab will introduce the state machine to implement control logic that takes several inputs to control several outputs. A win will flash the hex displays at the same time and a miss will flash the hex displays alternatingly.

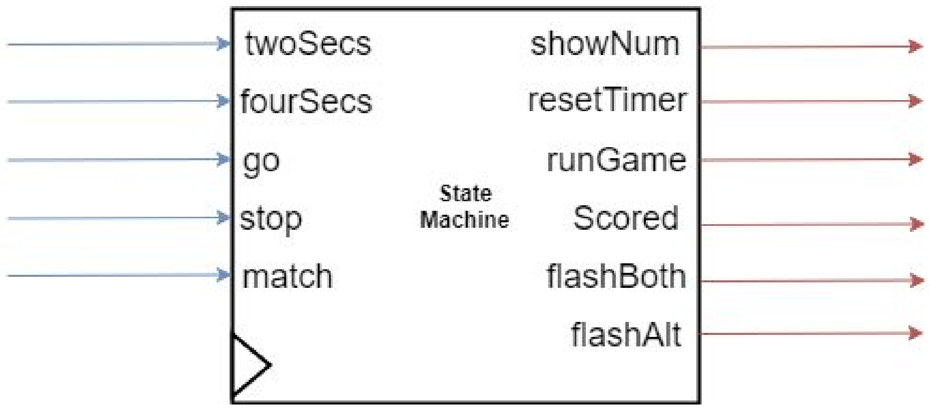
State Machine:
The state machine served as the brains for the entire game. Giving it several inputs, it routed through different states providing output variables at correct times, like runGame, which would run until stopped. Every important signal came from the state machine.

Method:
An altered state Machine derivation process was used because of one-hot encoding style flip flop management. There was no need to make a state transition table and derive logic though K-Maps. Starting the same way though, a state diagram was drawn out, and every input arrow was Next State Logic for each state.

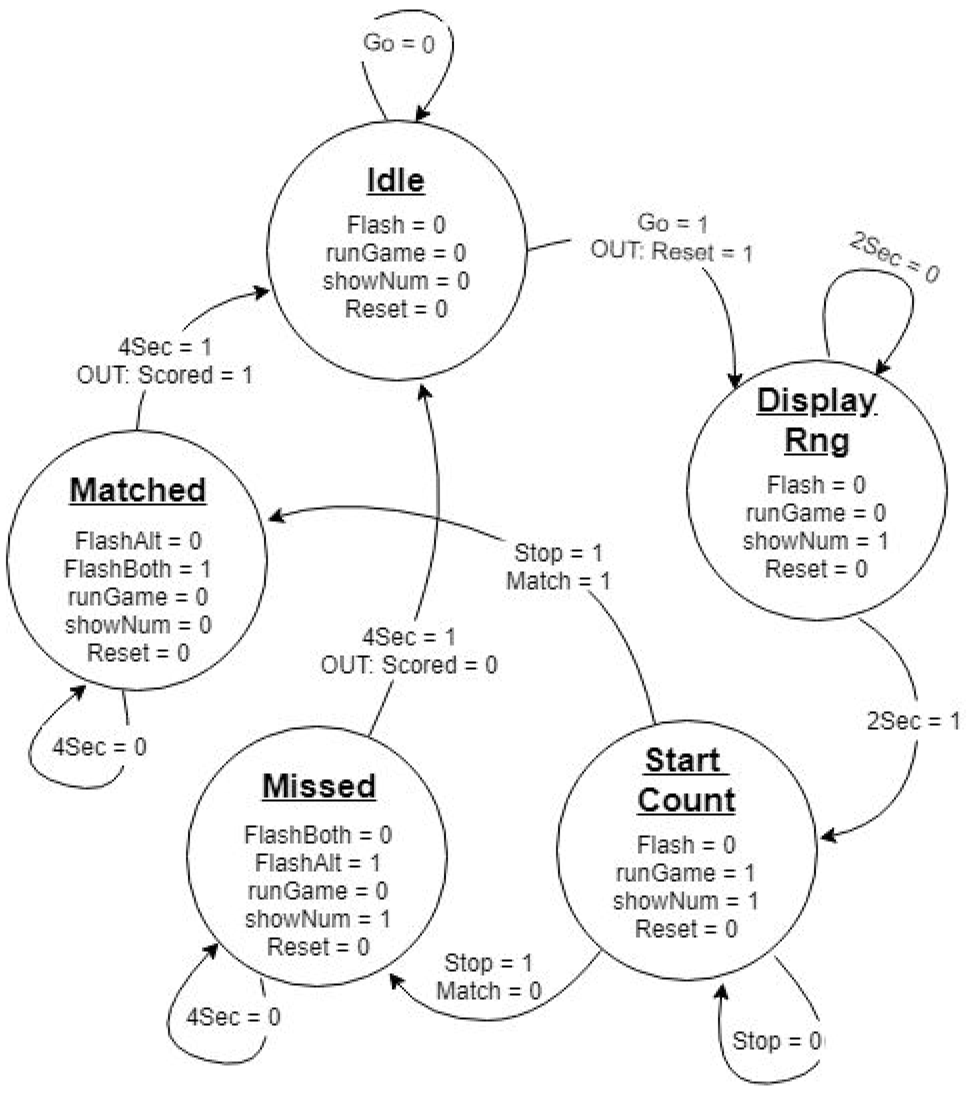
States:

- IDLE: The machine sat here until either go or stop was pressed. It held the game count value from the previous round on the right screen and the count on the random number on the left was hidden.
- Display RNG: When go is pressed, the random number is displayed on the left and then we sit in this state for two seconds.
- Start Count: The game counters counts up and resets every 0x3F. Wait here until stop is pressed.
- Matched: If stop is pressed on the correct number, sit here for four seconds flashing both screens. Return to Idle.
- Missed: If stop is pressed on the incorrect number, sit here for four seconds and alternating flash the hex displays. Return to idle.

Results:



The design began with creating a state diagram to account for every state and state path:



From this, the input and output logic of the states were derived. With one-hot encoding this became very simple:

Next State Logic:

- $D_{Idle} = 4sec \cdot Q_{match} + 4sec \cdot Q_{miss} + \sim Go \cdot Q_{idle}$
- $D_{Disp. Rng} = (Go \cdot Q_{Idle}) + (\sim 2Sec \cdot Q_{Disp. Rng})$
- $D_{Start} = (2sec \cdot Q_{disp. Rng}) + (\sim Stop \cdot Q_{start})$
- $D_{Miss} = (\sim 4sec \cdot Q_{Miss}) + (Stop \cdot \sim Match \cdot Q_{start})$
- $D_{Match} = (\sim 4sec \cdot Q_{Match}) + (Stop \cdot Match \cdot Q_{start})$

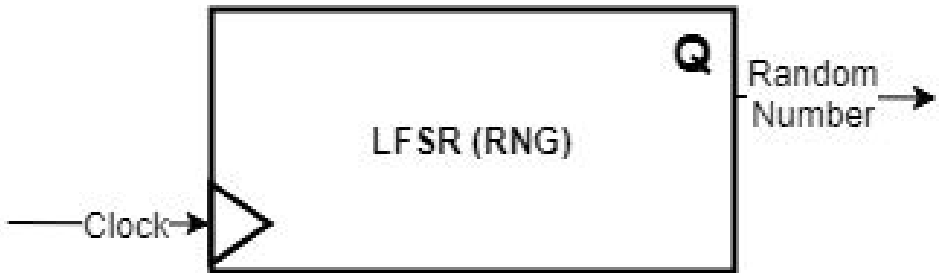
Outputs:
- showNum = $Q_{start} + Q_{Disp. Rng} + Q_{Match} + Q_{Miss}$
- runGame = Q_{Start}
- Reset = $Q_{Idle} \cdot Go$
- flashAlt = Q_{Miss}
- flashBoth = Q_{Match}

Random Number Generator :
The RNG cycles through all 255 possible bit combinations of an 8-bit number. When selecting outputs at varying intervals, the logic behaves like a random number generator. This was used to select a number to try to stop on. Only the bottom 6 bits were used because of time constraints.

Method:
The LFSR design was almost entirely articulated in the Lab description. Eight flip flops connected in series with a 4 input xor of bits 0, 5, 6, and 7 of the flip flop outputs. One of the flip flops had to be initialized to zero, as there was no actual input other than the clk.

A 6-bit register was put behind this LFSR to catch a number at a random time to hold. The LFSR was constantly cycling through random numbers and wouldn’t work unless there were flip flops to catch a value at a random time.

RESULTS:



The module had Inputs:
- Clock

And Outputs:
- Random Number

Simulation: The LFSR was outputting 0's at first and required one of the flip flops to initialize to 1.

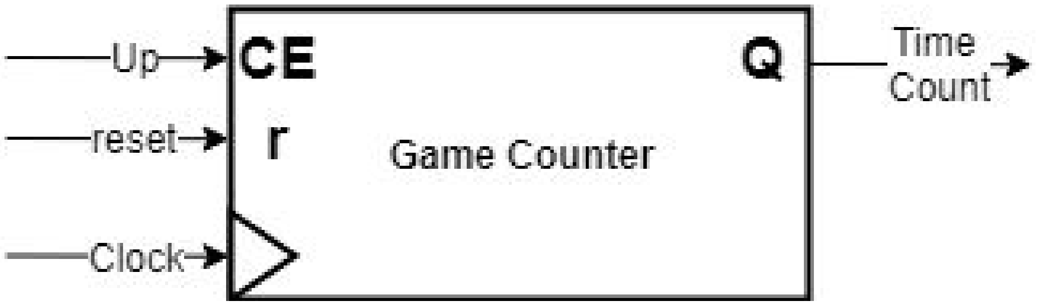
Game Counter:

The game counter counted up once every quarter second and was the right two displays of the game. Only 6 bits were used and displayed to the screen, and the counter would roll over to 0 at the 0x3F. The state machine sent a gameRun output to control when the game counter ran.

Method:

The design for the counter was grabbed from one of the previous labs that had up, down, and load inputs. The up input was connected to the gameRun output and the other two inputs was hard coded to zero. The reset for the counter was bound to the reset input or when the counter equaled 100000, one more than 0x3F, to get that roll over feature.

Results:



The module had Inputs:
- Clock (clk)
- Reset (r)
- Up Enable (CE)

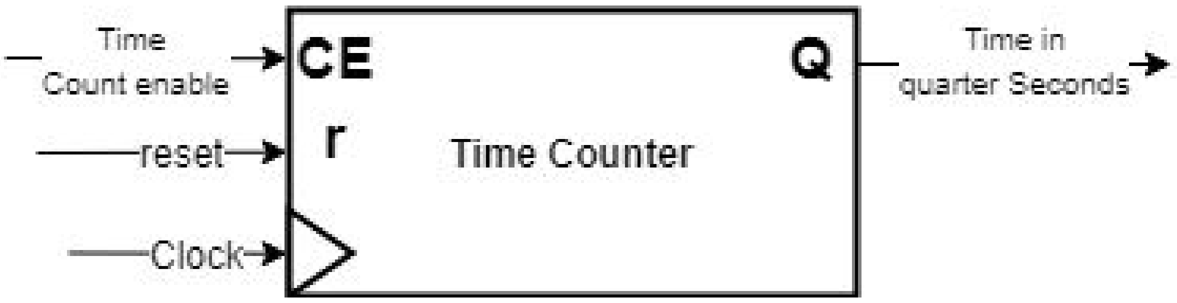
And Outputs:
- Game Time Count (Q[7:0])

ERROR: Interesting error within my counter was not setting the down and load control to 0 would break the entire counter.

Time Counter:
The time counter gave the signals for twoSec and fourSec.

Method:
Repurposing the counter from previous labs, the timeCounter took a clock input for synchronization, and a write enable that was tied to up count from our previous counter. The CE was set to signals timeCount AND qsec, as we wanted to keep track of quarter seconds to derive seconds from. By counting every quarter second, we could use more significant bits as indicators of two seconds and four seconds.

Results:



The module had Inputs:

- Clock (clk)
- Reset (r)
- Time Start (CE)

And Outputs:

- Time in quarter seconds (Q[7:0])

Simulation: The time counter was best tested when loaded by the actual board to use the correct clock speed versus using a simulated one which wasn't necessarily the board clock speed. The reset was tested, and just needed to start at 0 when pressed.

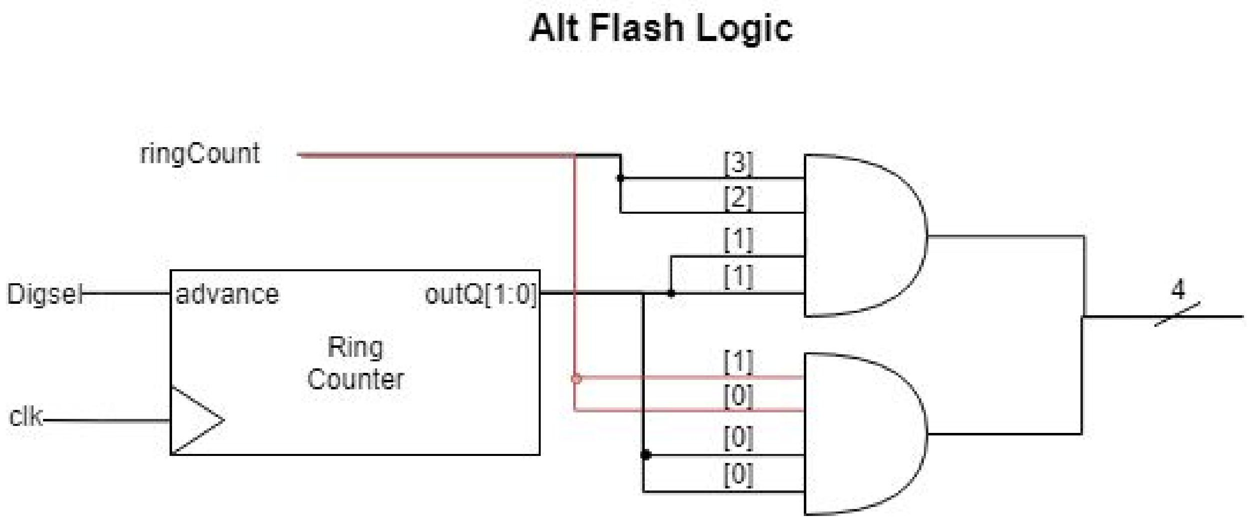
Display:

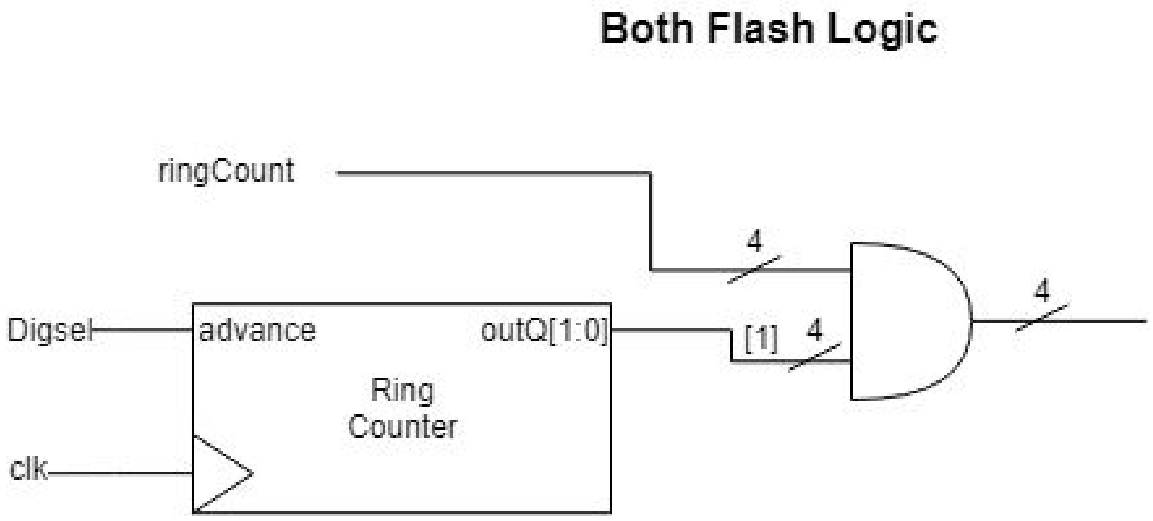
The display logic controlled the anode signals. This lab through some curveballs by making the game flash the anodes alternatingly and at the same time when either winning or losing the game for four seconds. The game also hid the left two displays when stuck in idle. which the state machine controlled with the inputs showNum, flashBoth, and flashAlt.

Method:

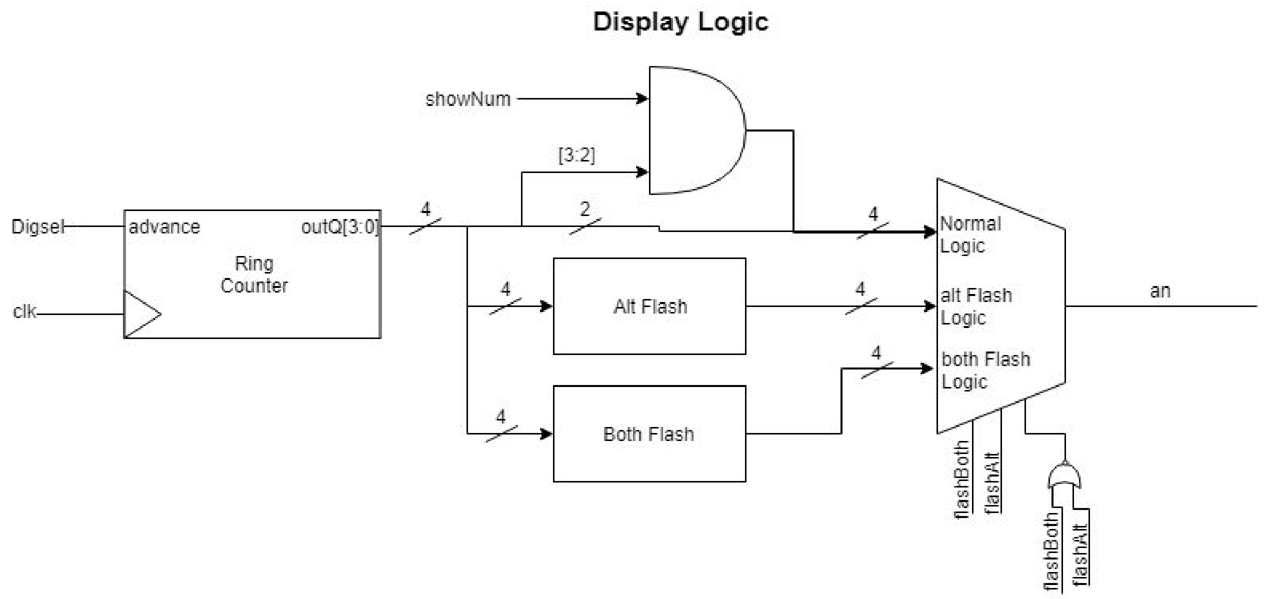
The design for this was by not necessarily the best way to do it, but is what worked best for me. Using a 3 to 1x4 mux, the three different inputs were four bits, and were the different logics for each mode of anode display logic. The displays were either doing there normal logic i.e. not flashing and controlling anode 3 and 4's of and on with a control showNum signal, or the displays were flashing alternatingly or at the same time. The flashing logic used a 2 bit ringcounter that took input qsec. Each of a bits was sign extended by 1 bit and then associated to the anodes.

Results:





The first input from the ring counter is sign extended to four bits.



Corrections: The Display logic wasn't necessarily the most straightforward, but worked best for me. Instead of using a 2-bit ring counter to flash the displays, the qsec signal could've been AND'ed with the anode enable lines to get the same flashing effect.

Simulation: Simulating tested how the flashBoth, flashAlt and showNum signals affected the enable wires of the anodes. The simulation file also used the state machine to make sure that the working state machine worked in tandem with the display logic to flash appropriately.

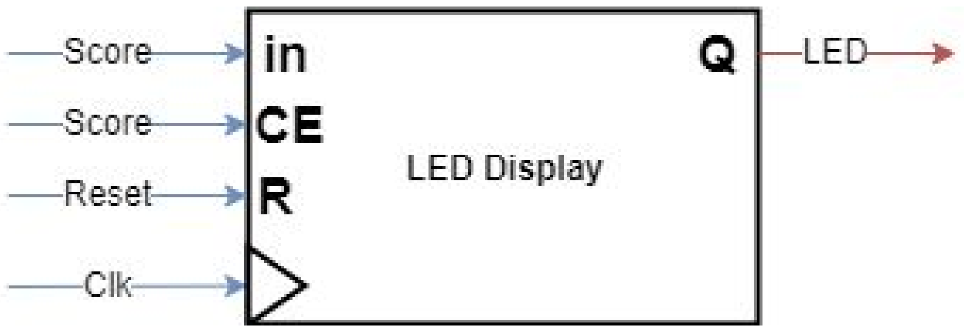
LED Display:

The LED Display uses the 16 LEDs on the basys board to display the score the user has, where each lit LED equals 1 point.

Method:

This score holder is a bit shift register. Each time a score is made, the new 1 bit shifts all in push all the bits forward 1. The difficulty is in making sure that the write enable doesn't cause this bit shift to happen for many clock cycles. The state machine's output for scoring a point is set on a transition between states so this mealy style output mitigates that issue. The design is a basic 16-bit shift register, connected in series.

Results:



Both the input data and write enable are tied to the state machine output Score. This assures that the bit shifter should receive no more than 1 clock cycles worth of bits.

Simulation: Testing this in simulation was focused on assuring the bit shifter didn't hold a score input for several cycles. This was paired with the state Machine in simulation to test this.

Top Level Design:

The top level connects everything together. The state machine is the heart of the entire circuit implementation, and almost all of the logic is either feeding the state machine signals or receiving signals from the state machine. The state machine cycles through 5 states providing the outputs for specified amounts of time and values. When starting each round it waits two seconds before counting up and when winning or losing, flashes the displays for four seconds.

Method:

The full assembly began with the state machine and the front end logic. Making sure the correct signals were getting sent to the state machine was most important to me.

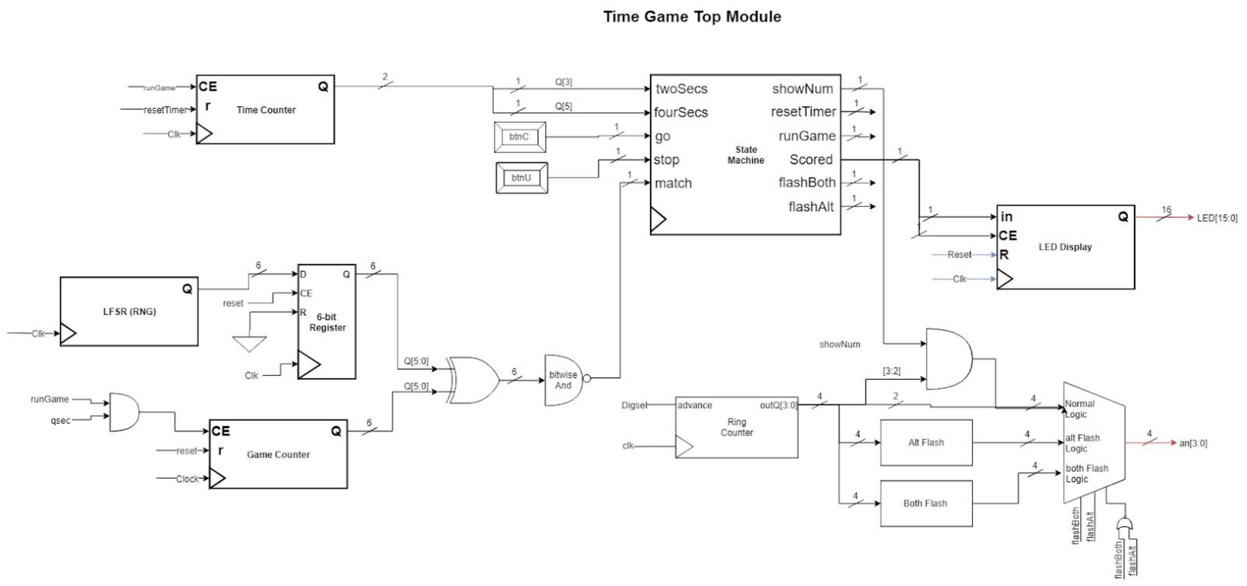
<u>Time Counter:</u> The two seconds and four seconds signal came from the time counter. With the time counter incrementing every quarter second, taking the third and fifth bit gave us two seconds and four seconds respectively. This also relied on the time counter resetting after certain states, otherwise those two seconds and four second bits would stay high and wouldn't give the values desired.

<u>Game Counter</u>: The match signal came from the LFSR and game counter. A reset signal went off when going from Idle to display RNG states. This signal enabled the write enable of the

6-bit register connected to the output of the LFSR, essentially catching another random value. This value was compared with the value stopped on from the game counter by XOR'ing them and then bitwise NAND. This outputs a 1 if the values are the same and a 0 if not.

The display logic just needed the correct signals connected to them and worked correctly.

Results:

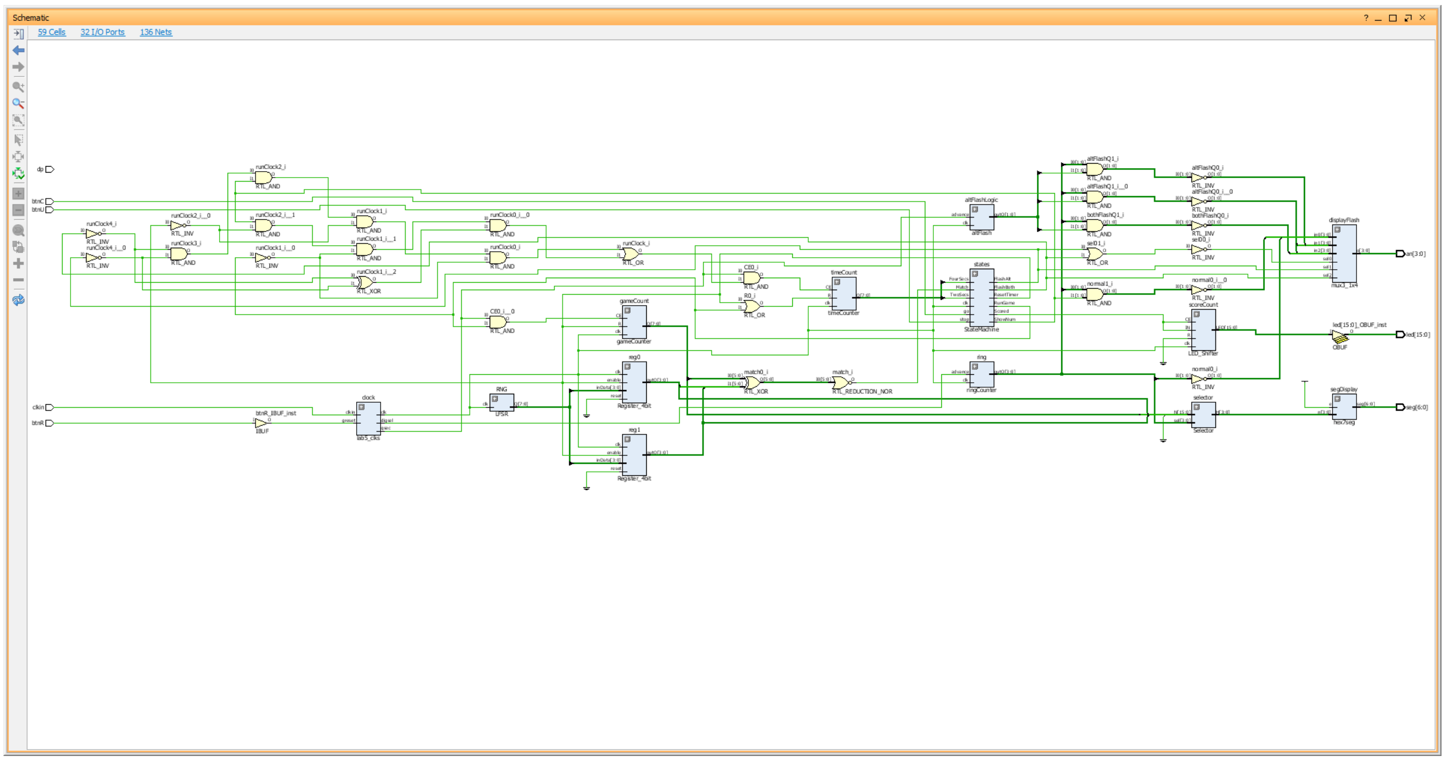


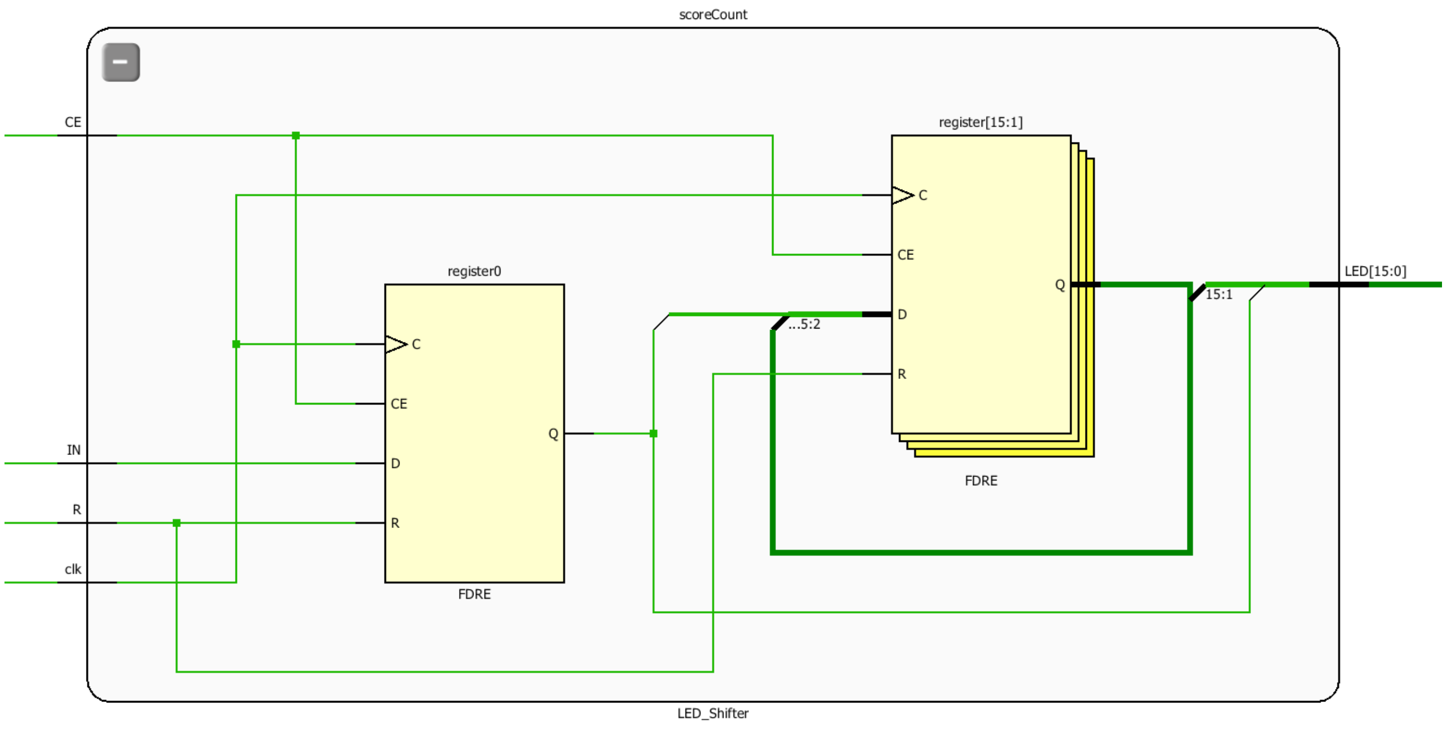
NOTE: The diagram excludes the notion of active low signals for the display logic for ease of schematic flow.

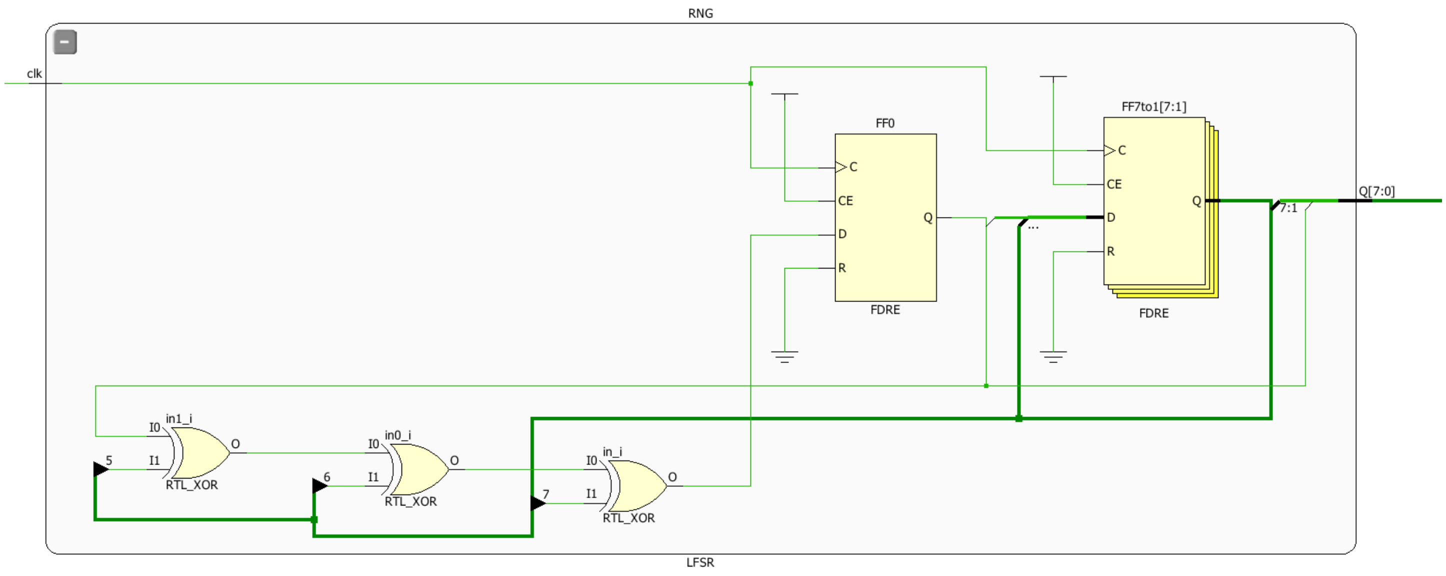
Simulation: Most of the issues for me in this lab came from flashing the displays. This was difficult to test in the simulation file, so most was done through implementation on the board.

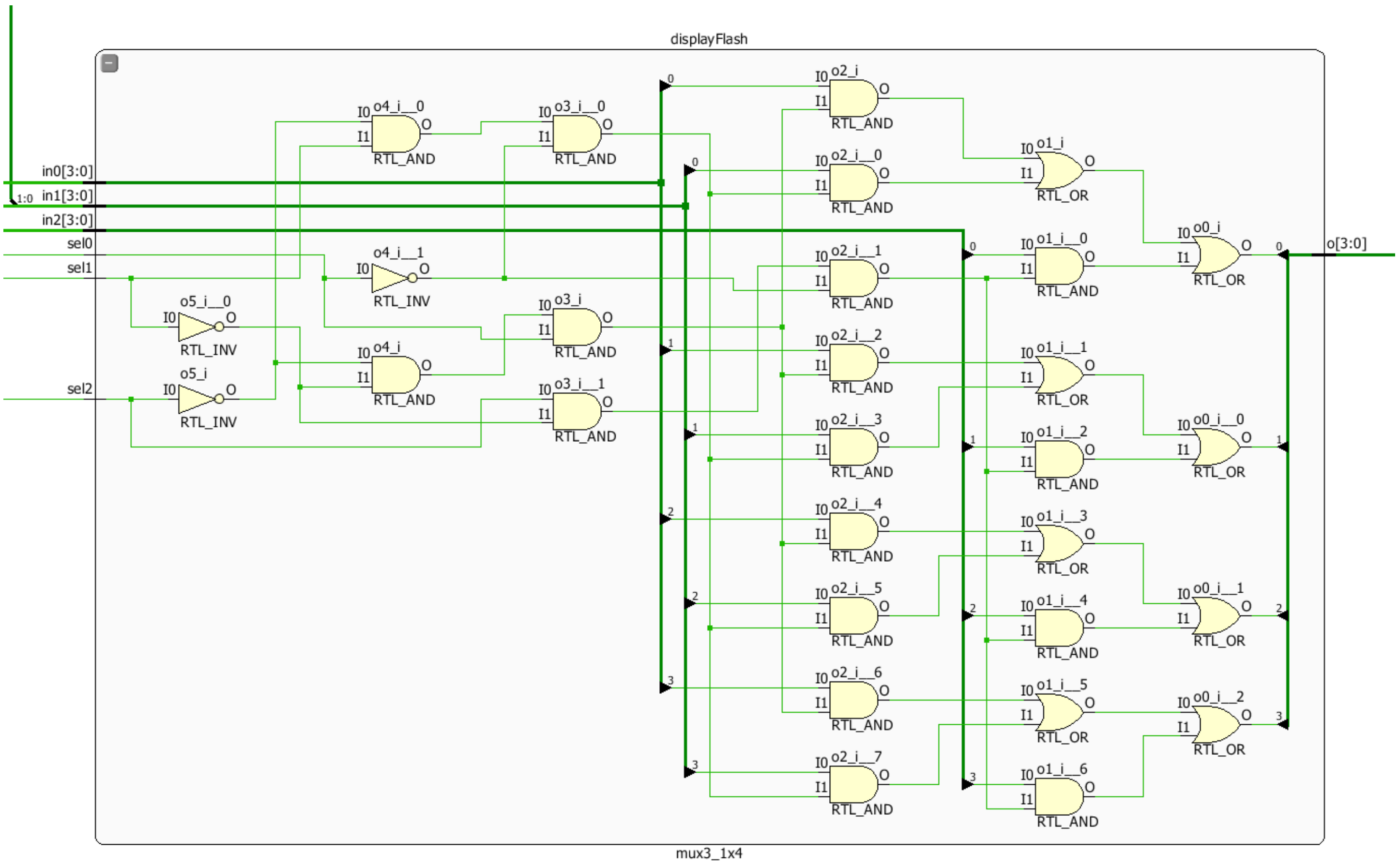
Conclusion

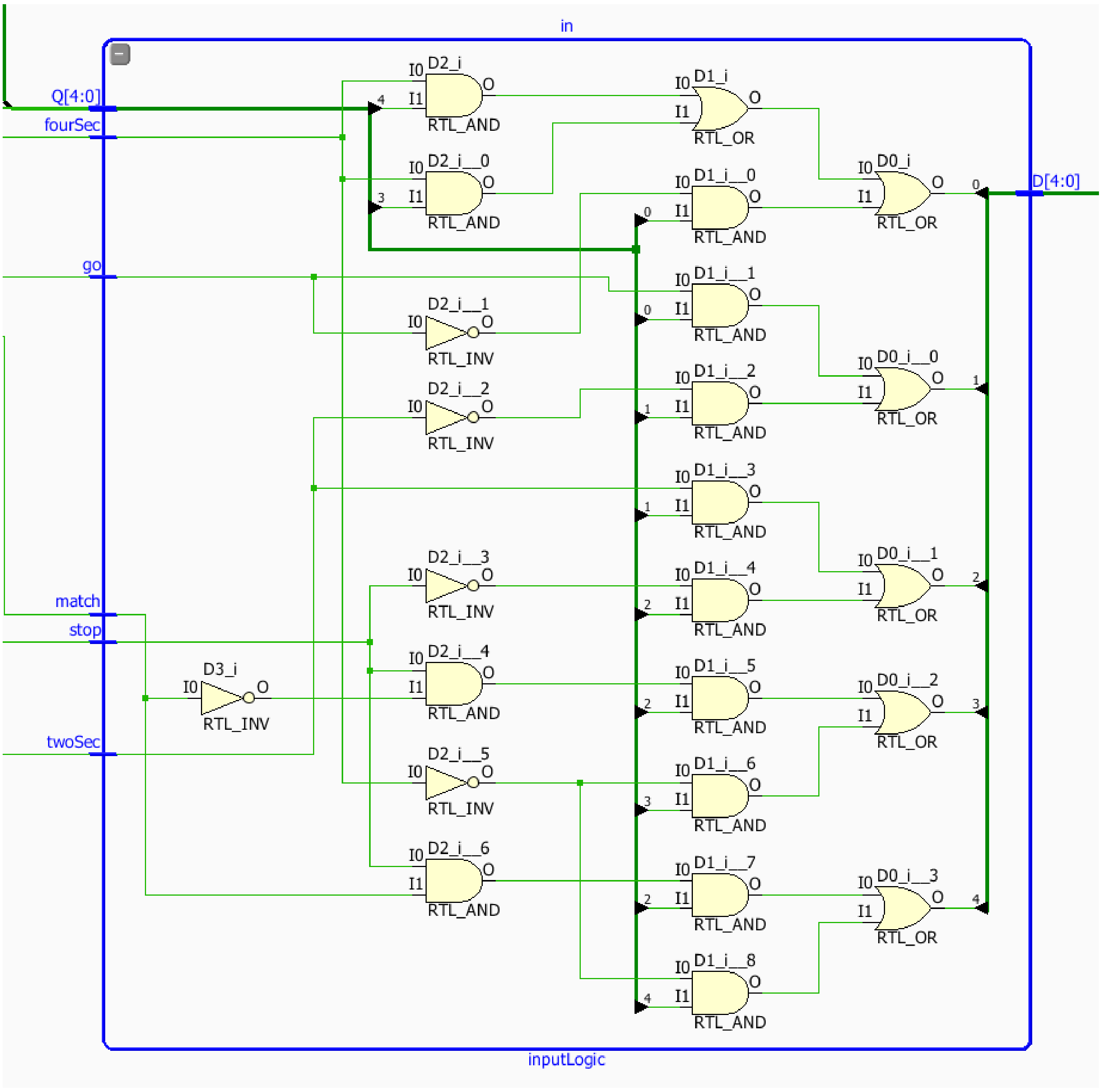
This lab taught the use of state machines to control logic. This style of module design is obviously very powerful and gives extreme control of outputting more nuanced data manipulation. If I were to recreate the implementation, I would do a better job with the anode logic and simplify it with just using the qsec signal from the provided lab clock module.

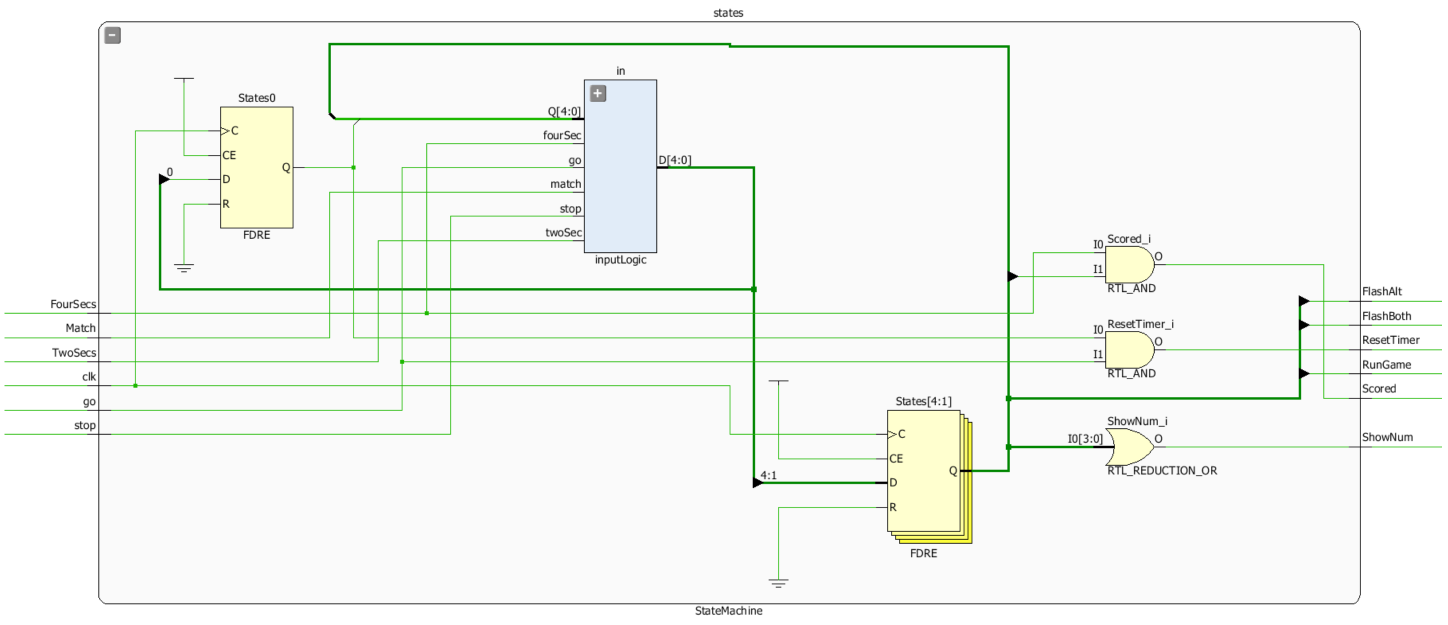

Schematic
59 Cells
32 I/O Ports
136 Nets
dp
btnC
btnU
clkin
btnR
runClock4_i
RTL_INV
runClock4_i__0
RTL_INV
runClock2_i__0
RTL_INV
runClock3_i
RTL_AND
runClock2_i
RTL_AND
runClock2_i__1
RTL_AND
runClock1_i__0
RTL_INV
runClock1_i
RTL_AND
runClock1_i__1
RTL_AND
runClock1_i__2
RTL_XOR
runClock0_i__0
RTL_AND
runClock0_i
RTL_AND
runClock_i
RTL_OR
CE0_i__0
RTL_AND
CE0_i
RTL_AND
R0_i
RTL_OR
timeCount
timeCounter
gameCount
gameCounter
reg0
Register_4bit
reg1
Register_4bit
RNG
LFSR
clock
lab5_clks
btnR_IBUF_inst
IBUF
match0_i
RTL_XOR
match_i
RTL_REDUCTION_NOR
altFlashLogic
altFlash
states
StateMachine
ring
ringCounter
altFlashQ1_i
RTL_AND
altFlashQ1_i__0
RTL_AND
bothFlashQ1_i
RTL_AND
sel01_i
RTL_OR
normal1_i
RTL_AND
altFlashQ0_i
RTL_INV
altFlashQ0_i__0
RTL_INV
bothFlashQ0_i
RTL_INV
sel00_i
RTL_INV
normal0_i__0
RTL_INV
normal0_i
RTL_INV
scoreCount
LED_Shifter
selector
Selector
displayFlash
mux3_1x4
led[15:0]_OBUF_inst
OBUF
segDisplay
hex7seg
an[3:0]
led[15:0]
seg[6:0]

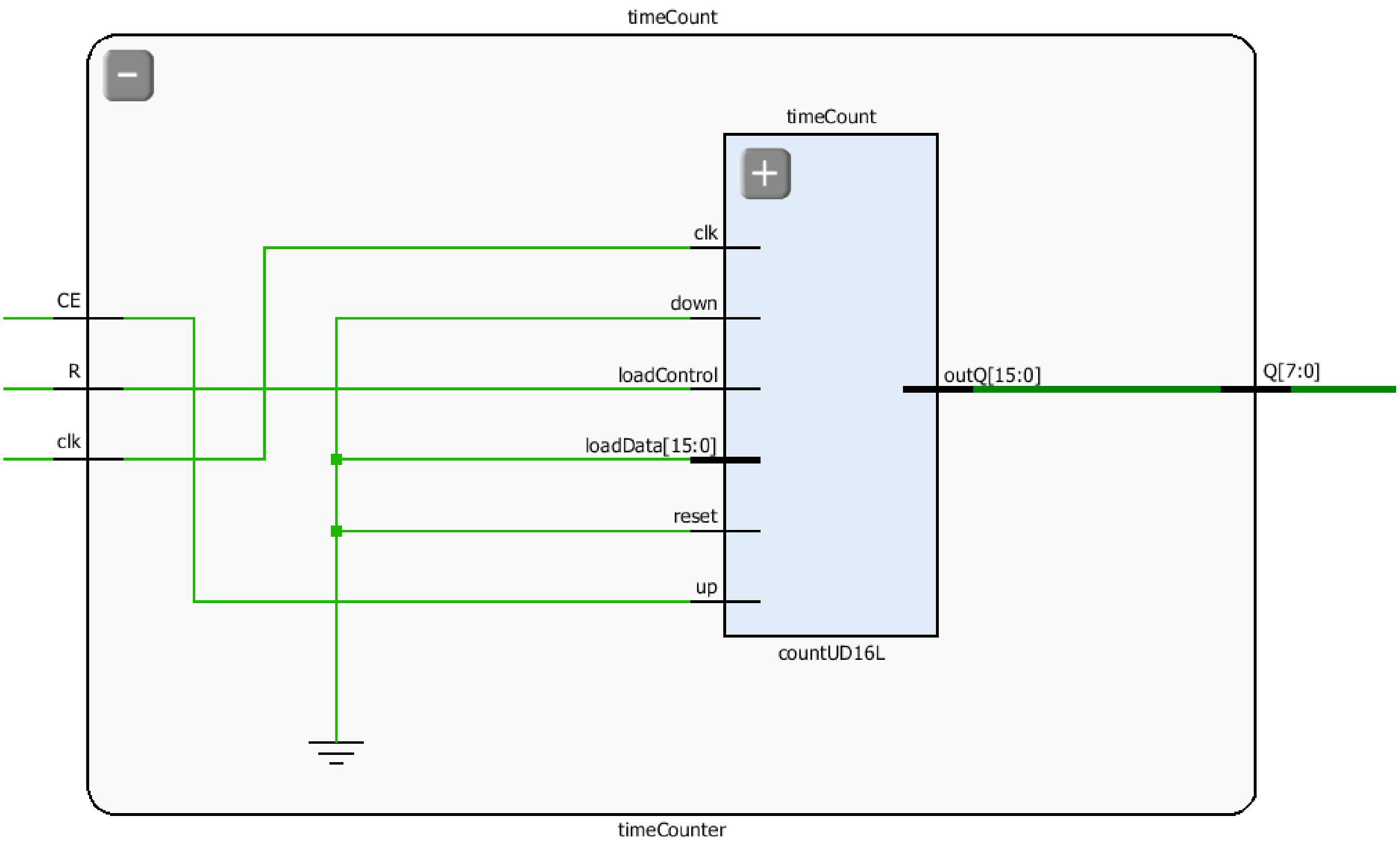

scoreCount
CE
register[15:1]
C
CE
Q
D
R
register0
C
CE
D
Q
R
IN
R
clk
FDRE
FDRE
...5:2
15:1
LED[15:0]
LED_Shifter

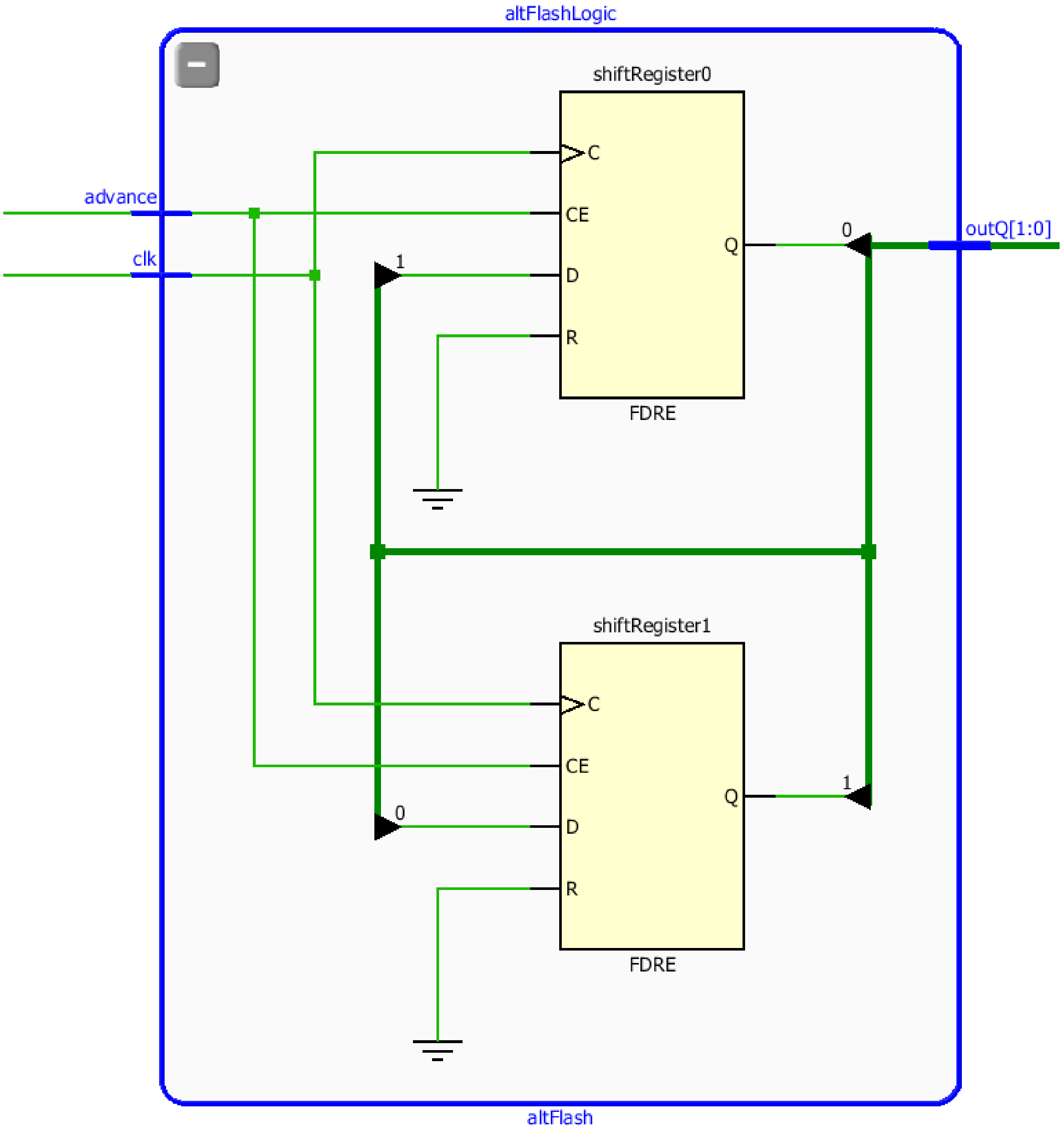

RNG
clk
FF0
C
CE
D
R
Q
FDRE
FF7to1[7:1]
C
CE
D
R
Q
FDRE
...
7:1
Q[7:0]
in1_i
I0
I1
O
RTL_XOR
5
in0_i
I0
I1
O
RTL_XOR
6
in_i
I0
I1
O
RTL_XOR
7
LFSR

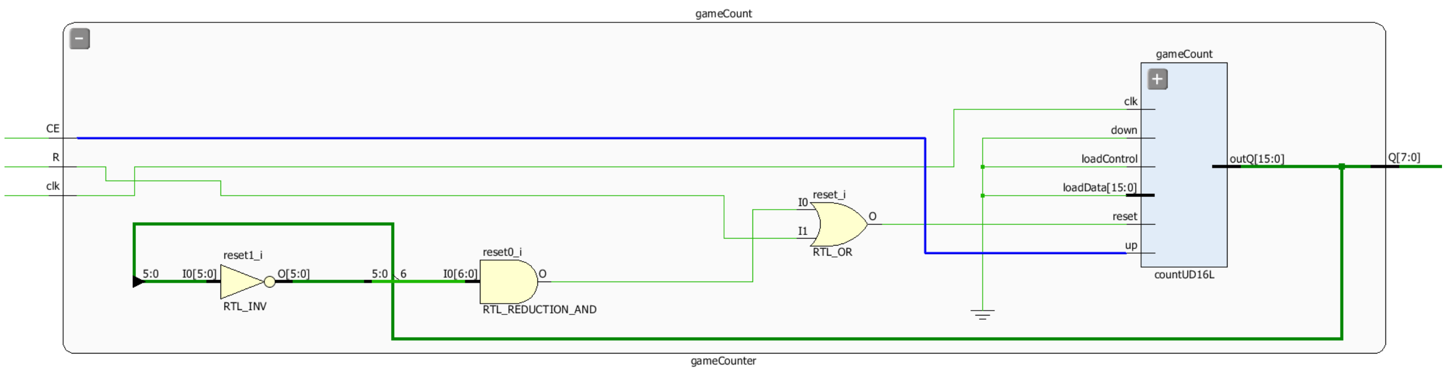

displayFlash
in0[3:0]
in1[3:0]
in2[3:0]
sel0
sel1
sel2
o4_i__0
o3_i__0
o4_i__1
o5_i__0
o5_i
o3_i
o4_i
o3_i__1
o2_i
o2_i__0
o2_i__1
o2_i__2
o2_i__3
o2_i__4
o2_i__5
o2_i__6
o2_i__7
o1_i
o1_i__0
o1_i__1
o1_i__2
o1_i__3
o1_i__4
o1_i__5
o1_i__6
o0_i
o0_i__0
o0_i__1
o0_i__2
RTL_AND
RTL_OR
RTL_INV
o[3:0]
mux3_1x4


in
Q[4:0]
fourSec
go
match
stop
twoSec
D[4:0]
inputLogic
D2_i
D2_i__0
D2_i__1
D2_i__2
D2_i__3
D2_i__4
D2_i__5
D2_i__6
D3_i
D1_i
D1_i__0
D1_i__1
D1_i__2
D1_i__3
D1_i__4
D1_i__5
D1_i__6
D1_i__7
D1_i__8
D0_i
D0_i__0
D0_i__1
D0_i__2
D0_i__3
RTL_AND
RTL_OR
RTL_INV


states
States0
C
CE
D
R
Q
FDRE
0
in
Q[4:0]
fourSec
go
match
stop
twoSec
D[4:0]
inputLogic
Scored_i
I0
I1
O
RTL_AND
ResetTimer_i
I0
I1
O
RTL_AND
ShowNum_i
I0[3:0]
O
RTL_REDUCTION_OR
States[4:1]
C
CE
D
R
Q
4:1
FDRE
FourSecs
Match
TwoSecs
clk
go
stop
FlashAlt
FlashBoth
ResetTimer
RunGame
Scored
ShowNum
StateMachine


timeCount
timeCount
clk
down
loadControl
loadData[15:0]
reset
up
countUD16L
CE
R
clk
outQ[15:0]
Q[7:0]
timeCounter


altFlashLogic
shiftRegister0
C
CE
D
R
Q
FDRE
advance
clk
1
0
outQ[1:0]
shiftRegister1
C
CE
D
R
Q
FDRE
0
1
altFlash


gameCount
CE
R
clk
gameCount
clk
down
loadControl
loadData[15:0]
reset
up
countUD16L
outQ[15:0]
Q[7:0]
reset_i
I0
I1
O
RTL_OR
reset1_i
5:0
I0[5:0]
O[5:0]
RTL_INV
5:0 , 6
reset0_i
I0[6:0]
O
RTL_REDUCTION_AND
gameCounter

```
`timescale 1ns / 1ps
//UP REGISTER
module Register_4bit(
    input [3:0] inData,
    input clk,
    input enable,
    input reset,
    output [3:0] outQ
    );

    FDRE #(.INIT(1'b0) ) register0(.C(clk), .R(reset), .CE(enable), .D(inData[0]), 
.Q(outQ[0]));
    FDRE #(.INIT(1'b0) ) register1(.C(clk), .R(reset), .CE(enable), .D(inData[1]), 
.Q(outQ[1]));
    FDRE #(.INIT(1'b0) ) register2(.C(clk), .R(reset), .CE(enable), .D(inData[2]), 
.Q(outQ[2]));
    FDRE #(.INIT(1'b0) ) register3(.C(clk), .R(reset), .CE(enable), .D(inData[3]), 
.Q(outQ[3]));

endmodule
```

```
`timescale 1ns / 1ps

module countUD16L(
    input clk, up, down, loadControl, reset,
    input [15:0] loadData,
    output upperLimit, lowerLimit,
    output [15:0] outQ
    );
    wire [3:0] p, q;

    assign lowerLimit = q[0] & q[1] & q[2] & q[3];
    assign upperLimit = p[0] & p[1] & p[2] & p[3];

    countUD4L count0(.clk(clk), .down(down),                      .lowerLimit(q[0]),
     .up(up),                      .upperLimit(p[0]),       .outQ(outQ[3:0]),
.loadData(loadData[3:0]),    .loadControl(loadControl), .reset(reset));
    countUD4L count1(.clk(clk), .down(down & q[0]),               .lowerLimit(q[1]),
     .up(up & p[0]),               .upperLimit(p[1]),       .outQ(outQ[7:4]),
.loadData(loadData[7:4]),    .loadControl(loadControl), .reset(reset));
    countUD4L count2(.clk(clk), .down(down & q[1] & q[0]),        .lowerLimit(q[2]),
     .up(up & p[1] & p[0]),        .upperLimit(p[2]),       .outQ(outQ[11:8]),
.loadData(loadData[11:8]),   .loadControl(loadControl), .reset(reset) );
    countUD4L count3(.clk(clk), .down(down & q[2] & q[1] & q[0]), .lowerLimit(q[3]),
     .up(up & p[2] & p[1] & p[0]), .upperLimit(p[3]),       .outQ(outQ[15:12]),
.loadData(loadData[15:12]),  .loadControl(loadControl), .reset(reset));

endmodule
```

```
`timescale 1ns / 1ps

module mux3_1x4(
    input [3:0] in0,
    input [3:0] in1,
    input [3:0] in2,
    input sel2, sel1, sel0,
    output [3:0] o
    );

     assign o[3] =  (in0[3] & (~sel2 & ~sel1 & sel0)) | (in1[3] & (~sel2 & sel1 &
~sel0)) | (in2[3] & (sel2 & ~sel1 & ~sel0));
     assign o[2] =  (in0[2] & (~sel2 & ~sel1 & sel0)) | (in1[2] & (~sel2 & sel1 &
~sel0)) | (in2[2] & (sel2 & ~sel1 & ~sel0));
     assign o[1] =  (in0[1] & (~sel2 & ~sel1 & sel0)) | (in1[1] & (~sel2 & sel1 &
~sel0)) | (in2[1] & (sel2 & ~sel1 & ~sel0));
     assign o[0] =  (in0[0] & (~sel2 & ~sel1 & sel0)) | (in1[0] & (~sel2 & sel1 &
~sel0)) | (in2[0] & (sel2 & ~sel1 & ~sel0));
endmodule
```

```
`timescale 1ns / 1ps

module altFlash(
    input clk, advance,
    output [1:0] outQ
    );
    wire [1:0] p;
    assign outQ = p;
    FDRE #(.INIT(1'b1) ) shiftRegister0(.C(clk), .R(reset), .CE(advance), .D(p[1]),
.Q(p[0]));
    FDRE #(.INIT(1'b0) ) shiftRegister1(.C(clk), .R(reset), .CE(advance), .D(p[0]),
.Q(p[1]));
endmodule
```

```
`timescale 1ns / 1ps

module LED_Shifter(
    input IN, CE, R, clk,
    output [15:0] LED
    );

    wire [15:0] p;

    assign LED = p;

    FDRE #(.INIT(1'b0) ) register0      (.C(clk), .R(R), .CE(CE), .D(IN), .Q(p[0]));
    FDRE #(.INIT(1'b0) ) register[15:1] (.C({15{clk}}), .R({15{R}}), .CE({15{CE}}),
.D(p[14:0]), .Q(p[15:1]));
endmodule
```

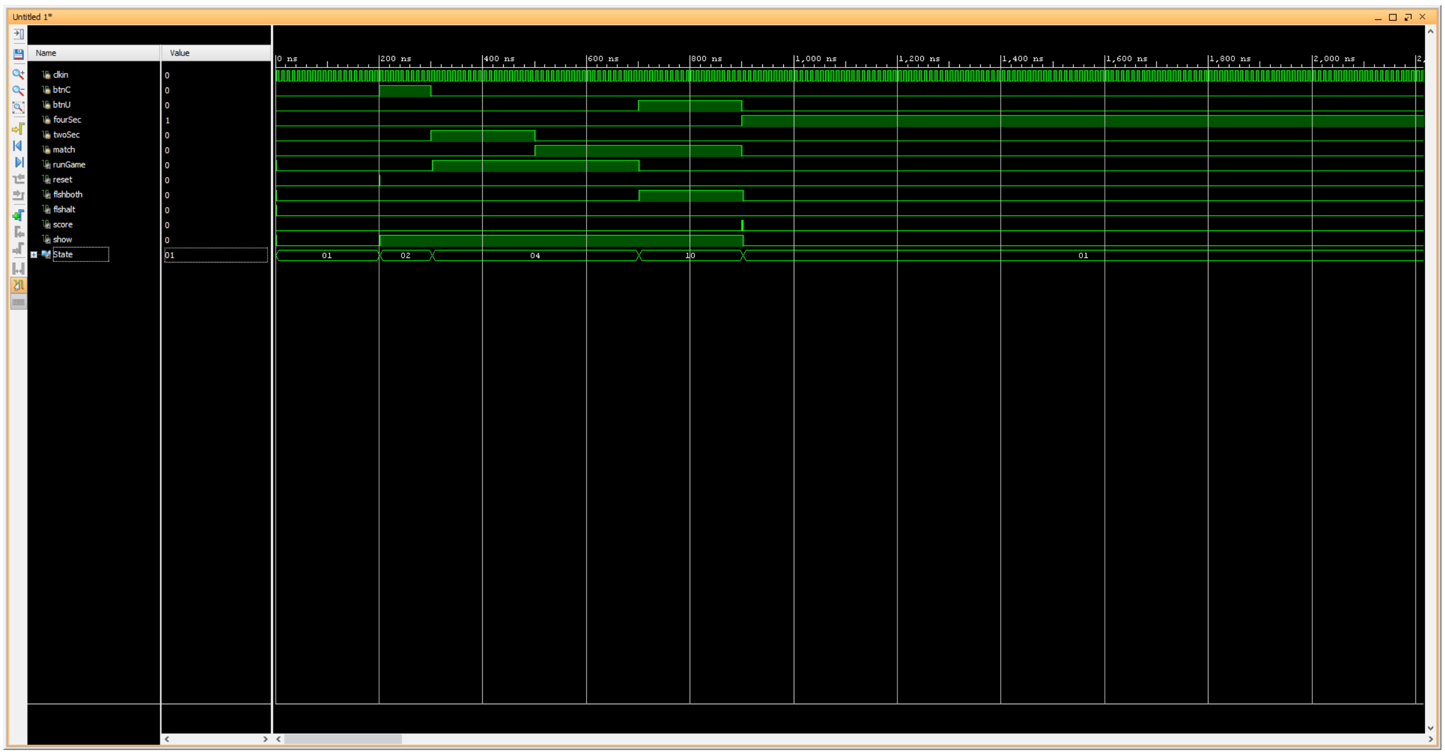
```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 02/12/2019 05:33:13 PM
// Design Name:
// Module Name: timeCounter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module timeCounter(
    input clk, R, CE,
    output [7:0] Q
    );


        countUD16L timeCount( .clk(clk), .up(CE), .down(1'b0), .loadControl(R),
                          .loadData({16'H0000}), .upperLimit(), .lowerLimit(),
.outQ({throwaway, Q}) );

endmodule
```

```
`timescale 1ns / 1ps


// This module recieves a random number from the RNG and holds it until its reset and
given a new
// RNG when CE is set to high.
module gameCounter(
    input clk, CE, R,
    output [7:0] Q
    );

   wire reset;

    // roll over to 0 at 0x3F or 111111
    assign reset = &{Q[6],~Q[5:0]} | R;

    wire [7:0] throwaway;
    countUD16L gameCount( .clk(clk), .up(CE), .down(1'b0), .loadControl(1'b0),
.reset(reset),
                          .loadData({16'H0000}), .upperLimit(), .lowerLimit(),
.outQ({throwaway, Q}) );


endmodule
```

```
`timescale 1ns / 1ps

module LFSR(
    input clk,
    output [7:0] Q
    );

    wire [7:0] rnd;

    assign Q = rnd;

    wire in;

    assign in = rnd[0]^rnd[5]^rnd[6]^rnd[7];

      FDRE #(.INIT(1'b0) ) FF7to1[7:1]         (.C( {7{clk}} ), .R({7{1'b0}}),
.CE({7{1'b1}}),   .D(rnd[6:0]),        .Q(rnd[7:1]));
      FDRE #(.INIT(1'b1) ) FF0                 (.C(    clk   ), .R(1'b0),
.CE(1'b1),        .D(in),                .Q(rnd[0]));

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 02/12/2019 06:27:04 PM
// Design Name:
// Module Name: inputLogic
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module inputLogic(
    input go, stop, twoSec, fourSec, match,
    input [4:0] Q,
    output [4:0] D
    );

    assign D[0] = (fourSec & Q[4]) | (fourSec & Q[3]) | (!go & Q[0]); //IDLE
    assign D[1] = (go & Q[0]) | (!twoSec & Q[1]);                      //Display RNG
    assign D[2] = (twoSec & Q[1]) | (!stop & Q[2]);                   //Start Counter
    assign D[3] = (stop & !match & Q[2]) | (!fourSec & Q[3]);         //Missed
    assign D[4] = (stop &  match & Q[2]) | (!fourSec & Q[4]);         //Matched
endmodule
```

```
`timescale 1ns / 1ps


module StateMachine(
    input go, stop, FourSecs, TwoSecs, Match, clk,
    output ShowNum, ResetTimer, RunGame, Scored, FlashBoth, FlashAlt
    );

    wire [4:0] Q, D;

    inputLogic in(.go(go), .stop(stop), .fourSec(FourSecs), .twoSec(TwoSecs),
.match(Match), .D(D), .Q(Q));


    FDRE #(.INIT(1'b1) ) States0 (.C(clk), .R(1'b0), .CE(1'b1), .D(D[0]), .Q(Q[0]));
    FDRE #(.INIT(1'b0) ) States[4:1] (.C({4{clk}}), .R({4{1'b0}}), .CE({4{1'b1}}),
.D(D[4:1]), .Q(Q[4:1]));

    assign ShowNum    = |{Q[4:1]};
    assign ResetTimer = Q[0] & go;
    assign RunGame    = Q[2];
    assign Scored     = FourSecs & Q[4];
    assign FlashBoth  = Q[4];
    assign FlashAlt   = Q[3];

endmodule
```


Untitled 1*

Name	Value
clkin	0
btnC	0
btnU	0
fourSec	1
twoSec	0
match	0
runGame	0
reset	0
fishboth	0
fishalt	0
score	0
show	0
State	01