Ramzey Ghanaim

The Lab 7 Write-Up

June 3, 2016

Description

The purpose of this lab was to use a VGA output on the Basyas2 board to create a game called Skyfall. In this game the user controls a bar at the bottom of the screen (the slug) which moves left or right. Push button 2 allows the slug to change its' direction. Also when the slug gets to the border of the playable space, the slug will change direction. While the player moves the slug back and forth, meteors fall from the sky and it is the player's job to avoid the meteors. The score is kept on the 7-segment display on the Basyas2 board. The left most digit contains the current level while the two rightmost digits keep track of the total number of metros that passed. The round score is kept as a hex value while the right two digits are displayed as decimal numbers.

Methods

To begin this project, I started designing something familiar: the state machine. My state machine was small with only five states. These states will be further discussed in the Results section. Once my state machine was completed I simulated it to work out any minor errors so once I connect the state machine with the VGA controller, I would not need to worry about the state machine having an issue. I was surprised to find my state machine worked perfectly the first time. Once the state machine was complete the next step was to design the horizontal and vertical syncs to allow the display to function to my needs.

Afterwards, I created the background with a module I called "Sync." This sync module is Verilog code which defines the region for the active region and edges of the screen. An active screen was need as a result of the way the board is communicating with the monitor. With Cathode Ray Tube (CRT) TVs the ray gun shoots electrons onto the screen one line at a time, taking time to transition from the end of one line to the front of the next. This CRT standard is how this lab is communicating with the monitor. Because of the time that it takes for the ray gun to move to the next line, I used an 800 x 525 grid, while the actual monitor size, and active region has a size of 640 x 480. The extra grid space allows time for the "ray gun" to move back from the end of one line to the beginning of the next. It was in the "Sync" where I defined the borders of the game and the border color, (more on this in the Results section).

Once the Sync was complete the next tasks were to create the slug and get it moving. After the slug was moving, I then moved to create metros and get them moving. Next, I linked the state

machine to the VGA controller which contains the Slug, Meteor, and Sync modules. I tested and debugged for endless hours before moving on the score board. The score, level and round were calculated in one module with three counters, one for each digit on the 7-segment display. The 7-segment display was the last component I needed to complete the game.

Results: State Machine

The state machine was the first component I designed for this lab. I came up with five states for the lab and they are as follows:

1. Menu: In many games there is a Main Menu where the game sits before the user pushes the proper buttons to start a game. My Menu state preforms the same function. Here, the game waits for the user to push pb0 to begin the game.
2. Meteor Falling: Once the player hits pb0 meteors start falling and the game begins. In this state, meteors keep on falling until one of three things happen: round reaches 10, a crash is detected, or the end of the game is reached.
3. Crash: If a crash is detected in the Meteor Falling state, the state machine will arrive here. In this state, the meteor and slug freeze and the slug flashes until the user hits pb1 to return to the Menu state.
4. Expand Slug: Once a round is complete (the user reaches a score of 10) the slug and meteor freeze for 8 seconds. During this time the slug flashes magenta for 4 seconds, then turns back to blue and expands. Once this happens the slug continues to flash for the remaining 4 seconds. Once the total 8 seconds are up, the game returns to the Meteor Falling state.
5. End Game: When the switches match the current round (leftmost digit on the display) and the player completes this round, the game will be over and the state machine will arrive in this state from the Meteor Falling state. In this phase, the meteors and slug freeze and a victory dance occurs where the slug and meteor flash and change colors.

Once I established these states, I came up with the state diagram that can be seen in Figure 1 on the next page.

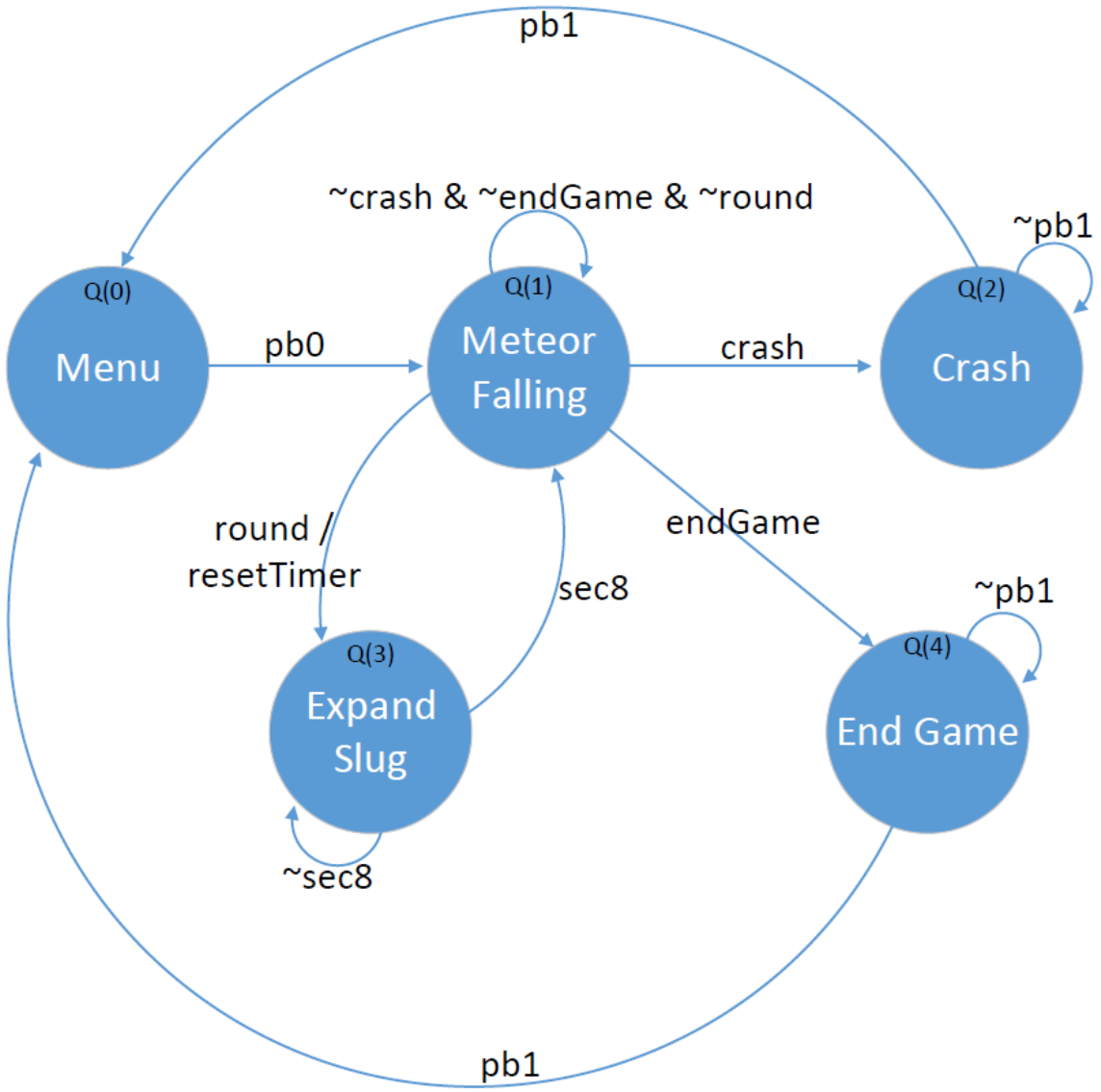


Figure 1: State Diagram for Sky Fall

After the state diagram was created, equations were made for each state and put into Verilog. The equations can be seen in Figure 2.

```
assign D[0] = Q[0]&~pb0 | Q[4]&pb1 | Q[2]&pb1;
assign D[1] = Q[1]&~crash&~endGame&~round | Q[0]&pb0 | Q[3]&sec8;
assign D[2] = Q[2]&~pb1 | Q[1]&crash&~round&~endGame; //crash state
assign D[3] = Q[3]&~sec8 | Q[1]&round&~crash&~endGame;
assign D[4] = Q[4]&~pb1 | Q[1]&endGame;
```

Figure 2: Verilog equations for the state diagram

I used one-hot encoding to encode the values for my state machine. The states and there encodings are as follows in Table 1 on the next page.

		Q(4)	Q(3)	Q(2)	Q(1)	Q(0)
End Game		1	0	0	0	0
Expand Slug		0	1	0	0	0
Crash		0	0	1	0	0
Meteor Falling		0	0	0	1	0
Menu		0	0	0	0	1

Table 1: One hot encoding table of the state machine

After the equations were written, the next step was to determine the outputs of the state machine. These outputs include resetting the timer (8 second timer), resetting the score, freezing the slug and meteors, victory dance, expand slug, and initialize everything when in the Menu state (Q0). My final equations for the outputs are in Figure 3.

```
assign resetTime =  Q[1]&round&~crash;
assign resetScore = Q[4]&pb1 | Q[2]&pb1;
assign flashAll = Q[4];//Game over, you win
assign flashSlug = Q[2] | Q[3];
assign freezeSlug = Q[2] | Q[4] | Q[3];
assign freezeM = Q[3] | Q[2] | Q[0] | Q[4];
assign victoryD = Q[4];
assign expand = Q[3];
assign initaleyes = Q[0] | Q[4]&pb1 | Q[2]&pb1;
```

Figure 3: State Machine output equations

Once the state machine was complete, I made this into a symbol and completed the higher level Finite State Machine which is the same process completed in previous labs where the current state determines the next state with D-Flip Flops. A simulation of my state machine can be seen in Figure 4.

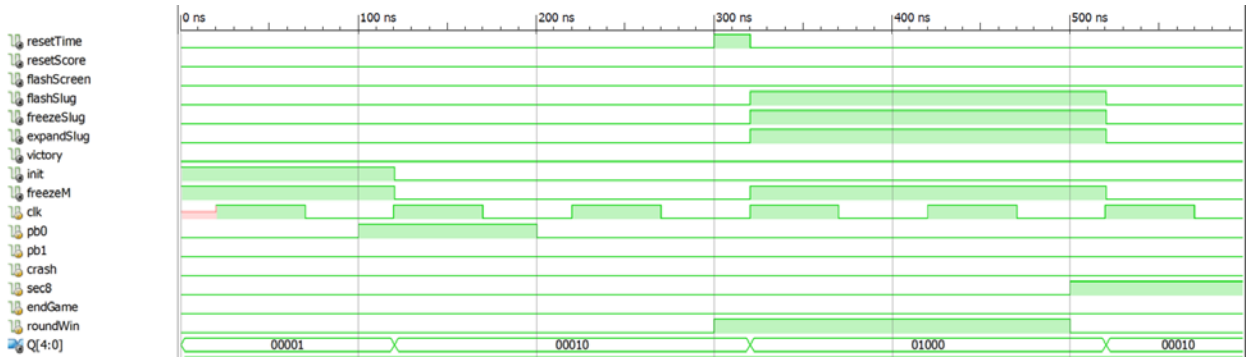


Figure 4: Simulation of the FSM

Results: Horizontal and Vertical Syncs

Once the state machine was completed and simulated correctly, the next step was to create the horizontal and vertical syncs to iterate through the screen. The iteration though the pixels has the exact same logic behind two nested for loops iterating through a 2D array. Once one row is complete the vertical counter is incremented by one and the horizontal counts all over again. To replicate nested for loops iterating through a 2D array, two 16-bit counters were used, one for the horizontal sync and the other for the vertical. The Horizontal sync needs to reset itself when it

gets to 800. This enables the vertical counter to increment by 1. Then the Horizontal (H) counter counts for the next row of pixels. Once the vertical counter reaches 525 and the horizontal counter reaches 800 the Vertical (V) counter is reset, completing one iterating through the screen, which is also called one frame. I called this the count in my schematic which is used later in the timer since about sixty frames occur in one second.

In the schematic, the HV control takes care of checking for 800 and 525 by ANDing the bits of the counters to detect the necessary numbers. The output is sent to the resets of the H and V counters. Figure 5 is the resulting schematic.

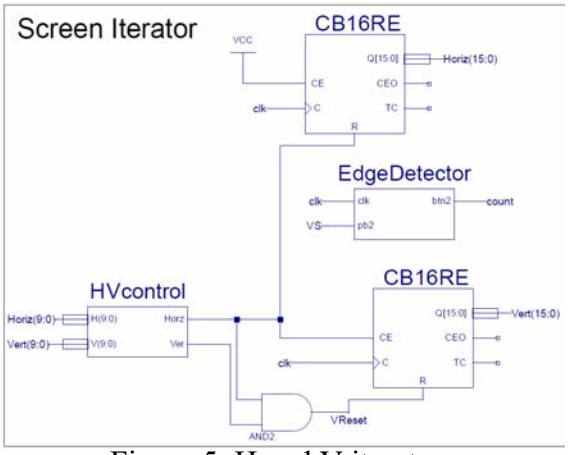


Figure 5: H and V iterators

A synchronizer is used to determine the active region and border of the game. The border uses 8 pixels on every side and the color is defined to be green. Each pixel takes 8 bits. Different combinations of 1's and 0's provide different colors. There are 3 bits that define green, 3 bits that define red, and two bits that define blue. The order of the bits from most significant to least is: G,G,G,R,R,R,B,B. Normally the R and G bits would be flipped, however I accidentally reversed the green and red bits in my obuf outputs and did not want to replace it since it does not matter, as long as I stayed constant through the rest of the project.

There are HS and VS signals are used by the monitor to synchronize the beginning of each row and frame. H-sync is low between the horizontal values of 655 and 751 while V-Sync is low between the vertical values of 490 and 491. The resulting Verilog code of the Sync module can be seen in Figure 6.

```
wire ActiveRegion = ((H <= 639) & (V <= 479));
wire RightEdge = ((H >= 632) & (H <= 639));
wire LeftEdge = ((H >= 0) & (H <= 7));
wire TopEdge = ((V >= 0) & (V <= 7));
wire BottomEdge = ((V >= 472) & (V <= 479));

assign HS = ~((H < 655) | (H > 751)); // 655 751
assign VS = ~((V < 490) | (V > 491)); //good

assign G2 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
assign G1 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
assign G0 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
assign R2 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
assign R1 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
assign R0 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
```
Figure 6: Synchronizer code

The color outputs (RBG_values) are sent to the VGA_RBG module to be ORed with the colors from the slug and meteor to get one output that is sent to the screen. The VGA_RBG controller was a very simple Verilog logic which can be seen in Figure 7.

```
assign RBG_out = RBG_values | Slug_RBG | Meteor_RBG;
```

Figure 7: Determining the final RBG output

Results: Slug

After the screen outputs were complete, it was time to put elements onto the screen. I started with the slug. I started by defining a left border and a right border for the slug, which contained the horizontal pixels of the border: 0-8 for the left border and 632-639. Next, a slug region was defined. The slug region is a rectangle with a height of 16 pixels and a starting with of 64. Every level the slug size increases by 16 pixels. The slug was designed for it to grow by 8 pixels on both sides. The specific size is determined by the current level. Since the level starts at 0, the base case size of slug is 64. This default size is determined by the numbers 72 and 8 in Figure 8. The level is multiplied by 8 and added to the right side of the slug and subtracted from the left side. The subtraction is necessary to move 8 pixels left. While adding is used to move to the right. Next I added code to change direction to go right when the slug region and the left border are both high, meaning the slug is inside or touching the border. Change direction left variable goes high when the slug region is high and the right border is high. This means I want the direction to be left. Lastly, the slug output that gets sent to the VGA_RBG controller is defined as high when slug region is high. Later on, when I was designing the victory dance, I came back and added code to change the color of the slug when the end of the game is reached. My final result can be seen in Figure 8.

```
assign LeftBorder = (H<=8 & H>=0);
assign RightBorder = (H>=632 & H<=639);



assign SlugRegion = (V <= 420 & V >= 404) & (H <= (72 + (level*8) + loc) & H >= (8- (level*8) + loc));

assign ChangeDirectionR =  SlugRegion&LeftBorder;
assign ChangeDirectionL =  SlugRegion&RightBorder;

//Slug_RBG_output is in the code bellow
assign slug_out [1] = (SlugRegion&~changeColor | SlugRegion&changeColor&flashes[5])&~endGame;
assign slug_out [0] = (SlugRegion&~changeColor | SlugRegion&changeColor&flashes[5])&~endGame;
assign slug_out [2] = SlugRegion&changeColor&flashes[5]&~sec4&~endGame | SlugRegion&endGame&flashes[4];
assign slug_out [3] = SlugRegion&changeColor&flashes[5]&~sec4 | SlugRegion&endGame&flashes[4];
assign slug_out [4] = SlugRegion&changeColor&flashes[5]&~sec4 | SlugRegion&endGame&flashes[4];
assign slug_out [5] = SlugRegion&endGame&flashes[4];
assign slug_out [6] = SlugRegion&endGame&flashes[4];
assign slug_out [7] = SlugRegion&endGame&flashes[4];
```

Figure 8: Slug

Results: Slug Movement

The slug moves based on the “loc” input in the line that describes the slug region on Figure 8. The location is determined by a counter which counts up or down based on whether the slug is moving right or left. I used a 16 bit CLED binary counter from the Xilinx library. Where the UP input is 1 if we want to count up (go right) or 0 if we want to count down (go left). I used logic to determine the UP state based on the changeDirectionL (L) and changeDirectionR (R) from the code in Figure 8. The logic works as follows: if L is high and R is not, we want to output 1 to the

counter to move right. This means we have hit the left border and want to go right. This is ORed with the logic to determine the direction with the push button (pb2). The PB is ANDed with the NOT of the current direction to switch it. The OR gate is then fed into a F-D flip flop and sent to the UP input on the counter.

During the menu state, we want the slug to spawn on the left of the screen, when the counter is at 0. Zero is loaded in the beginning of the menu state, since the initialize variable "init" will be high. This comes from the state machine as the "initialeyes" variable. It is high while we are in the menu state, I needed to edge detect init before sending it to the load input for the counter. Since we want the slug to move at one pixel per frame, I ANDed the count signal with the NOT of freeze. Freeze goes high when we want the slug to stop moving. The count signal is the signal generated by the edge detector in Figure 5 to count one frame. I should have made this signal name "frame" from the beginning as advised in the manual; however, I did not believe it was necessary, as long as I was constant. The output of the counter is sent to the slug Verilog code as the Move signal in Figure 9. The final logic can be seen in Figure 9.

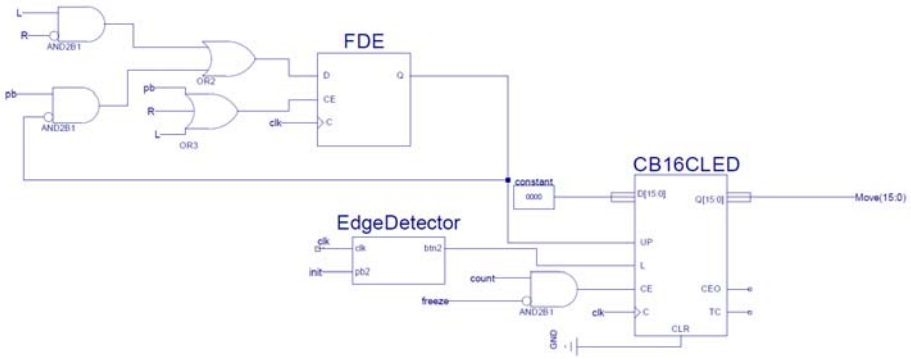


Figure 9: Moving the Slug

The logic demining the slug movement took a long time to fix. I first used a toggle flip flop to toggle the direction if the push button is pressed or the slug hits left or right border. My problem occurred in the event where the slug grows into the border, the slug becomes stuck. After trying for many hours to fix it by modifying what was defined as the edge and the logic that went into a toggle flip flop, I decided to not use a toggle flip flop. I used a standard FDE flip flop instead that is seen in Figure 9.

Results: Slug Expand

In order to expand the slug four seconds have had to pass. To make this work, I had a TC for 4 seconds that was sent as a load signal to load the new level into registers. The value in the registers is sent to the level signal inside the slug Verilog code. If I did not have these registers and sent the current level immediately, then the slug would grow instantly at the beginning of the Expand state rather than in the middle.

Results: Meteor

To create a meteor in the shape of a diamond we needed the formula: $|x - H| + |y - V| < 64$, where x and y are the coordinates for the center of the diamond and H and V are the H and V signals discussed earlier. Two diamonds are needed for this project so I needed to replicate this formula twice. Diamonds need to spawn above the active region so the y value for Meteor 1 is -40 while the y value for meteor 2 is -90. Now we needed to add the location of the diamond which is determined by the counters as the signal loc1 and loc2 for meteors 1 and 2 respectively. These loc values are multiplied by 2, since we want the meteor to move at 2 pixels per frame, and then added to the y-V piece of the formula. Computing x-H is similar; however, the x values come as a random generated number from the LFSR used in the previous labs. Two LFSRs are used, one for each meteor. So instead of x-H, I have lfsr1- H and lfsr2-H.

Once the subtraction was complete I had four values one x and one y for two meteors. From here these four values were sent outside the Meteor Verilog module to their own sign changers from previous labs. The sign changers compute the absolute value. The result is then sent back into the Meteor Verilog module.

Form here, I add the x and y values from the sign changers and check that they are less than 64 as the formula entails. I then created two additional meteors for meteor 1, doing the exact same thing but added 10 to the x and y values of the first additional meteor and added 20 to the x and y values of the second additional meteor. These additional meteors were created for both meteor 1 and meteor 2. It was necessary to do this to give the meteors a pattern. Once this was done I had three meteors for meteor 1 and three meteors for meteor 2. Each of the three meteors were sent to separate green bits for metor1's final RBG output. The same followed for meteor 2 except the color was assigned to the red bits. Later, when I added the end of game logic I came back and added some color changes for when the game gets to the end of game. On a final note of generating the position of the diamonds, if the meteor was the first in the game, I had logic telling it to spawn with respect the height in the middle of the screen.

Lastly, I had two load wires that went high when we wanted to reset the counter. Through various experimentation I found I wanted to reset the first diamond when the loc (counter) of the second diamond is at 200. I also discovered that I want to reset meteor 2 when the loc of the first diamond is at 150. One more signal was added to go high when the location of the meteors pass the screen. Figure 10 shows my Verilog code.

```
47     assign M1_outx = 264+lfsr1 - H;
48     assign M2_outx = (lfsr2+80 - H)&~firstMeteor;
49     assign M1_outy = (-40 - V + (loc1*2))&~firstMeteor | ((600 - V)+loc1*2)&firstMeteor ;
50     assign M2_outy = (-90 - V  + (loc2*2))&~firstMeteor;
51
52      //the formula:
53       wire meteor_1 = (M1_inx  + M1_iny < 64) ;
54       wire meteor_12 = (M1_inx +10 + M1_iny + 10) <64;
55       wire meteor_13 = (M1_inx + 20 + M1_iny + 20) < 64;
56       wire meteor_2 = (M2_inx  + M2_iny < 64);
57       wire meteor_22 = (M2_inx + 10 + M2_iny + 10) < 64;
58       wire meteor_23 = (M2_inx + 20 + M2_iny + 20) < 64;
59 //bellow meteor is green
60     assign meteor1[7] = meteor_1&~endGame;
61     assign meteor1[6] = meteor_12&~endGame;
62     assign meteor1[5] = meteor_13&~endGame;
63     assign meteor1[4] = meteor_12&endGame&flashes[4];
64     assign meteor1[3] = meteor_1&endGame&flashes[3];
65     assign meteor1[2] = meteor_13&endGame&flashes[4] |  meteor_1&endGame&flashes[3];
66     assign meteor1[1] = meteor_12&endGame&~flashes[4];
67     // bellow meteor is red
68     assign meteor2[2] = meteor_23;
69     assign meteor2[3] = meteor_22;
70      assign meteor2[4] = meteor_2;
71      assign meteor2[7] = meteor_23;
72
73      assign load1 = (loc2 == 200);// was at 200
74      assign load2 =  (loc1 == 150); // was 150
75    assign point = (loc1 >= 280) | (loc2 >= 320); //was 280 and 320
```

Figure 10: Meteor Verilog

Results: Meteor Counters

I used two 16 bit counters to move the meteors, where the meteor counters are loaded with 0 when the load1 and load2 signals go high from Figure 10, lines 73-74. To determine what height to load for the first meteor of the game, I had to use a mux to tell the meteor to spawn in the middle if we are initializing or spawn at 0 otherwise. I also had a logic to determine when to stop counting, when I want the meteors to freeze. This freeze input comes from the state machine. My results for the mux and the counters can be found in Figures 11 and 12.

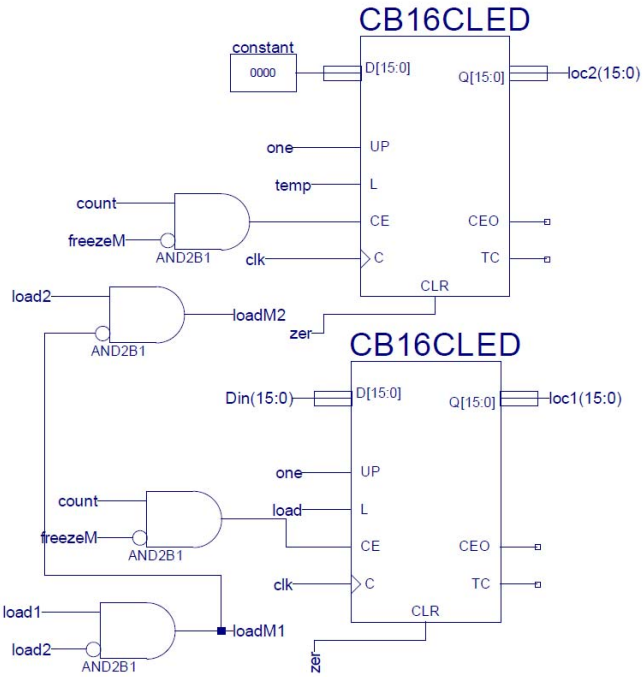


Figure 11: Meteor Counters

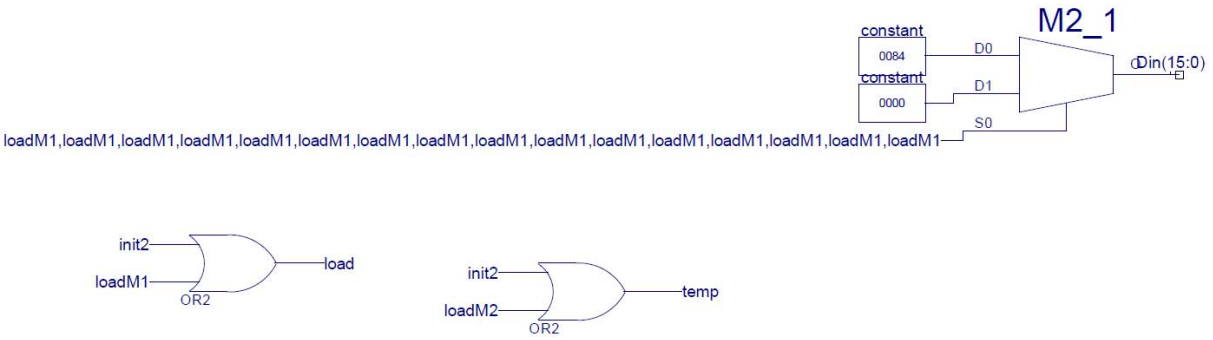


Figure 12: Logic determine loads for the counters

Lastly, the RGB for meteors 1 and 2 are ORed together before being sent to the VGA_RBG module in Figure 7.

Results: Score/7-Segment Display

Calculating the score was fairly simple. The score is the right most digit and the round is on the next one over. Both of these digits are counted in decimal so two decimal counters were used. The score increments every time a meteor passes the bottom of the screen. The point signal comes from the Verilog meteor code. This signal enables the score to increment. When the TC of the score goes high, the round increments, along with the level (Left most digit on the display). The level counter loads 1 as its first round. And continues increments from here whenever the score TC goes high.

The end of game is also detected here. When the switches match the level and the score matches, meaning the end of the switches' round is complete the game is over. Figure 13 and 14 shows my design.

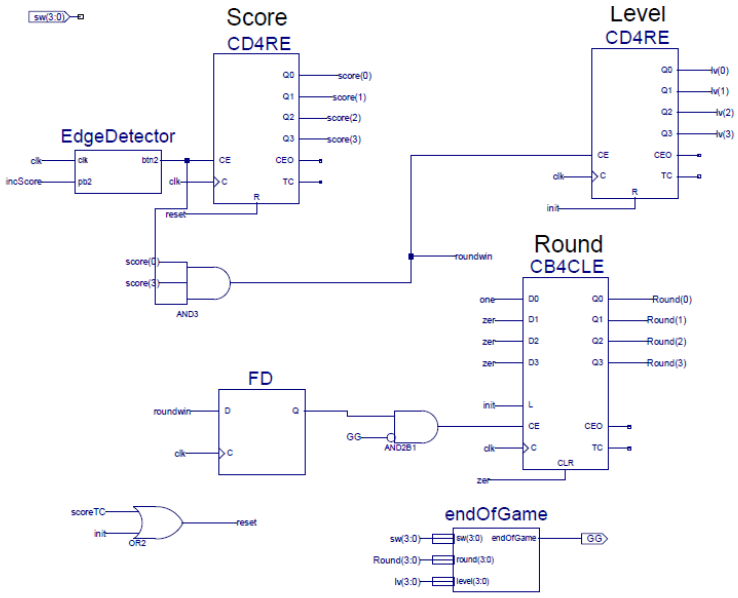


Figure 13: Score, Level, and Round calculator

My logic determining the end of the game is written in Verilog and can be viewed in Figure 14. When the signal goes high, the state machine goes to the end of game state where the victory colors for the slug and meteor activate.

```
assign endOfGame = (sw == round) & (level == sw);
```

Figure 14: Ending the Game detection

The Ring counter, hex7sex, and selector are all the same from previous labs. The Score, Level, and Round are sent to the display this display logic to display all the digits.

Results: Timer and Flasher

The Timer is the same as the previous timers in lab. When the counter gets to 511, the sec8 signal goes high, indicating 8 seconds have passed. When the counter gets to 255 the sec4 signal goes high. I am using the same count signal as the counter enable. For the timer. I call the counter value flash because I use the 5th bit to flash the slug and meteors when necessary. The 5th bit made a good strobe to flash the slug and meteors. My logic behind the timers can be seen I Figure 15.

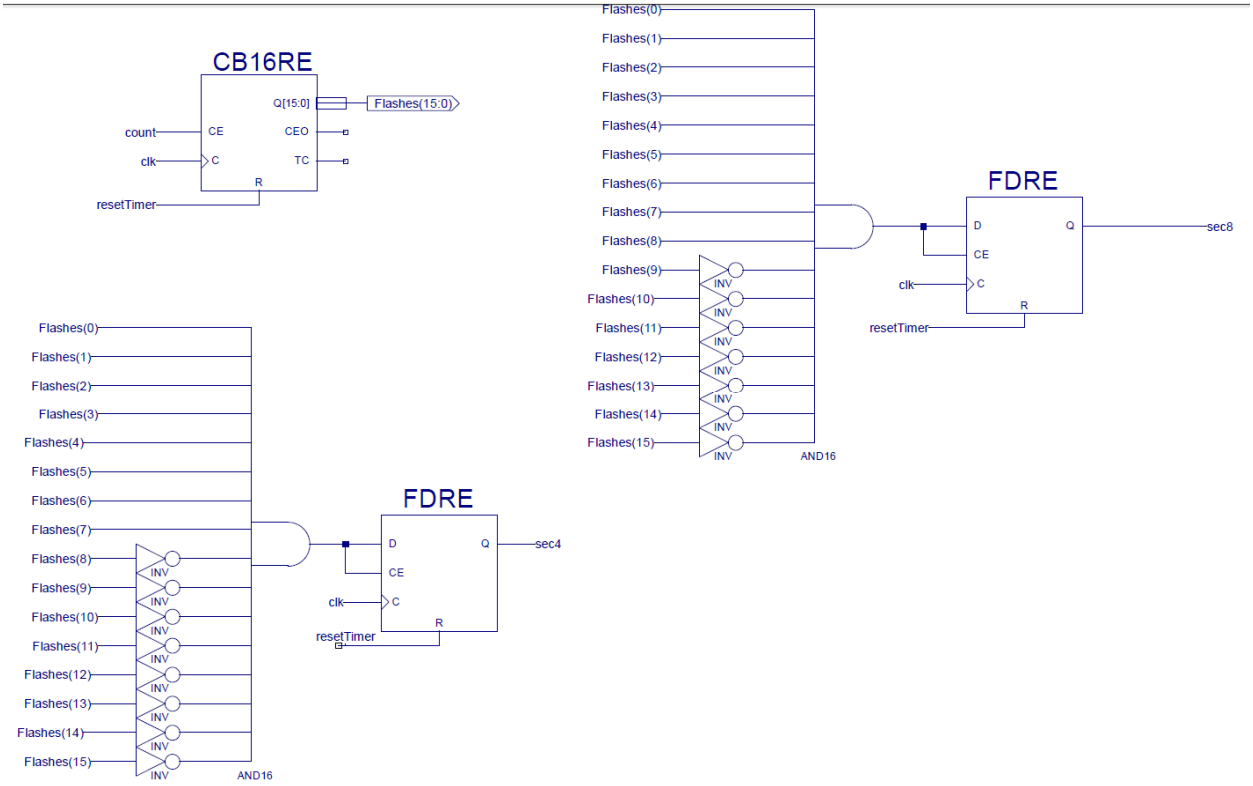


Figure 15: Timer and Flasher

Results: Crash Detection

The crash detection was very simple to implement. It only required one of the green meteor 1 bits to be ANDed with one of the blue bits from the slug, and one of the red meteor 2 bits to be ANDed with one of the blue bits from the slug. Since I combined the two meteors into on signal

(M) all I had to do was AND two bits, one red and one green with a blue bit from the slug. My final logic can be seen in Figure 16.

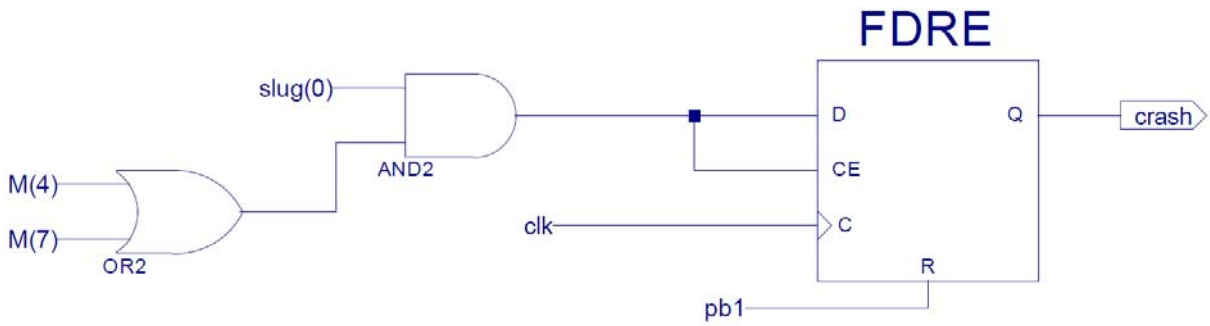


Figure 16: Crash Detection

Results: Top

On the top level, I created the proper clock as specified in the manual, created all the I/O bufs, and connected the state machine to the VGA controller. I was having some screen issues with the way pixels were coloring the screen, almost like ghosting. I put flip flops on all the RGB outputs that go to the display, allowing one clock cycle for all the bits to "catch up" so the ghosting is eliminated.

Results: Lab Questions

After generating a timing report, I found my clock can run at 32 MHz. I found the maximum time an input has to be set up is 10.768 ns while the longest hold time came to be -1.148 ns.

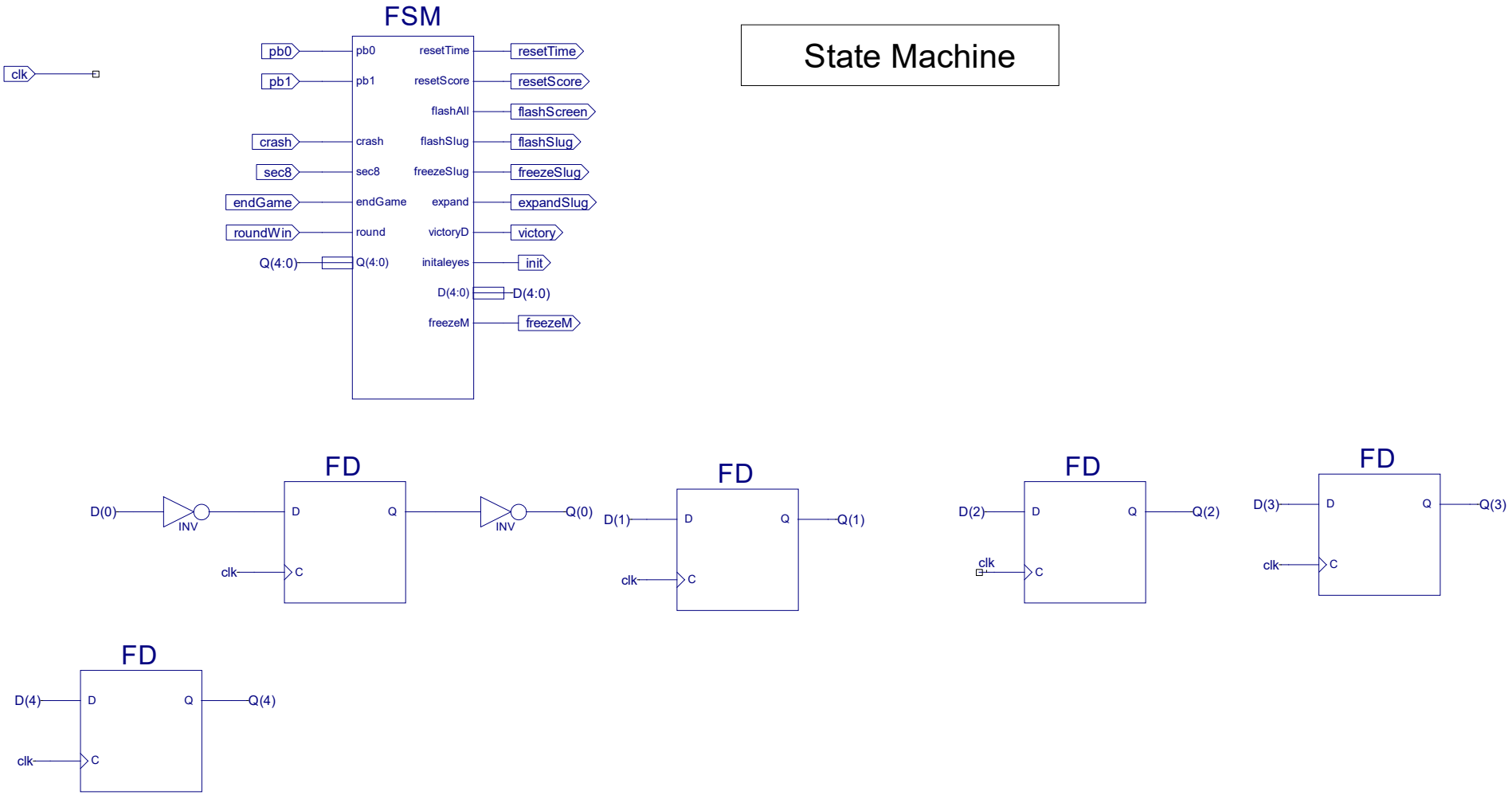
Conclusion

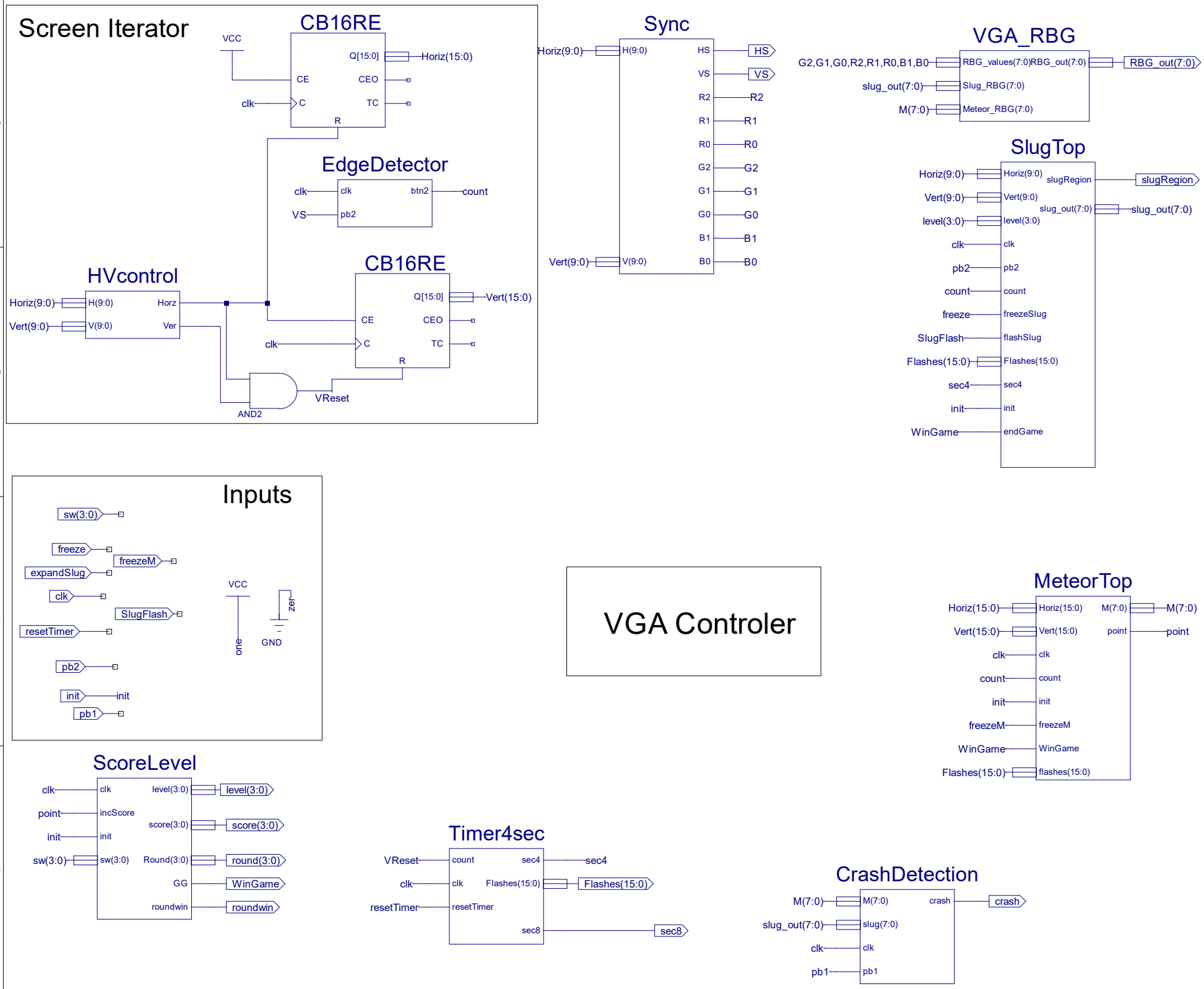
In this lab, I learned how to interact with a monitor over a VGA connection using 8 bit color depth. The process seemed intimidating, having to program each pixel with logic gates, however the horizontal and vertical syncs and their counters helped to automate the process of iterating through each pixel. If I were to redo the lab, I would have not used a toggle flip flop for the slug from the beginning instead of spending endless hours debugging issues over moving the slug. The one component I would like to optimize would be the sign changers, as it seems that sending inputs from the meteor, to the sign changer and back into the meteor module can definitely be simplified.

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    15:44:42 05/13/2016
7   // Design Name:
8   // Module Name:    FSM
9   // Project Name:
10  // Target Devices:
11  // Tool versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module FSM(
22     input pb0,
23     input pb1,
24     //input pb2,
25     input crash,
26     input sec8,
27     input endGame, //high when sw = round+1
28     input round, //high when ready to go to next round
29     input [4:0] Q,
30
31     output resetTime,
32     output resetScore,
33     output flashAll,
34     output flashSlug,
35     output freezeSlug, //high when we want to freeze frame
36     output expand, //high when in expand phase
37     output victoryD, //high when victory dance
38     output [4:0] D,
39     output initaleyes,
40     output freezeM
41
42      );
43
44     assign D[0] = Q[0]&~pb0 | Q[4]&pb1 | Q[2]&pb1;
45     assign D[1] = Q[1]&~crash&~endGame&~round | Q[0]&pb0 | Q[3]&sec8;
46     assign D[2] = Q[2]&~pb1 | Q[1]&crash&~round&~endGame; //crash state
47     assign D[3] = Q[3]&~sec8 | Q[1]&round&~crash&~endGame;
48     assign D[4] = Q[4]&~pb1 | Q[1]&endGame;
49
50     assign resetTime =  Q[1]&round&~crash;
51     assign resetScore = Q[4]&pb1 | Q[2]&pb1;
52     assign flashAll = Q[4];//Game over, you win
53     assign flashSlug = Q[2] | Q[3];
54     assign freezeSlug = Q[2] | Q[4] | Q[3];
55     assign freezeM = Q[3] | Q[2] | Q[0] | Q[4];
56     assign victoryD = Q[4];
57     assign expand = Q[3];
58     assign initaleyes = Q[0] | Q[4]&pb1 | Q[2]&pb1;
```

```
59
60    endmodule
61
```

State Machine




Screen Iterator
CB16RE
VCC
Q[15:0]
Horiz(15:0)
CE
CEO
clk
C
TC
R
EdgeDetector
clk
clk
btn2
count
VS
pb2
HVcontrol
Horiz(9:0)
H(9:0)
Horz
Vert(9:0)
V(9:0)
Ver
CB16RE
Q[15:0]
Vert(15:0)
CE
CEO
clk
C
TC
R
VReset
AND2
Sync
Horiz(9:0)
H(9:0)
HS
VS
R2
R1
R0
G2
G1
G0
B1
B0
Vert(9:0)
V(9:0)
VGA_RBG
G2,G1,G0,R2,R1,R0,B1,B0
RBG_values(7:0)
RBG_out(7:0)
RBG_out(7:0)
slug_out(7:0)
Slug_RBG(7:0)
M(7:0)
Meteor_RBG(7:0)
SlugTop
Horiz(9:0)
Horiz(9:0)
slugRegion
slugRegion
Vert(9:0)
Vert(9:0)
slug_out(7:0)
slug_out(7:0)
level(3:0)
level(3:0)
clk
clk
pb2
pb2
count
count
freeze
freezeSlug
SlugFlash
flashSlug
Flashes(15:0)
Flashes(15:0)
sec4
sec4
init
init
WinGame
endGame
Inputs
sw(3:0)
freeze
freezeM
expandSlug
clk
SlugFlash
resetTimer
pb2
init
init
pb1
VCC
one
GND
zer
VGA Controler
MeteorTop
Horiz(15:0)
Horiz(15:0)
M(7:0)
M(7:0)
Vert(15:0)
Vert(15:0)
point
point
clk
clk
count
count
init
init
freezeM
freezeM
WinGame
WinGame
Flashes(15:0)
flashes(15:0)
ScoreLevel
clk
clk
level(3:0)
level(3:0)
point
incScore
score(3:0)
score(3:0)
init
init
sw(3:0)
sw(3:0)
Round(3:0)
round(3:0)
GG
WinGame
roundwin
roundwin
Timer4sec
VReset
count
sec4
sec4
clk
clk
Flashes(15:0)
Flashes(15:0)
resetTimer
resetTimer
sec8
sec8
CrashDetection
M(7:0)
M(7:0)
crash
crash
slug_out(7:0)
slug(7:0)
clk
clk
pb1
pb1

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    16:10:25 05/14/2016
7   // Design Name:
8   // Module Name:    Sync
9   // Project Name:
10  // Target DeV[9:0]ices:
11  // Tool V[9:0]ersions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // ReV[9:0]ision:
17  // ReV[9:0]ision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module Sync(
22      input [9:0] H,
23      input [9:0] V,
24      output HS,
25      output VS,
26      output R2,
27      output R1,
28      output R0,
29      output G2,
30      output G1,
31      output G0,
32      output B1,
33      output B0
34      );
35
36      wire ActiveRegion = ((H <= 639) & (V <= 479));
37      wire RightEdge = ((H >= 632) & (H <= 639));
38      wire LeftEdge = ((H >= 0) & (H <= 7));
39      wire TopEdge = ((V >= 0) & (V <= 7));
40      wire BottomEdge = ((V >= 472) & (V <= 479));
41
42      assign HS = ~((H < 655) | (H > 751)); // 655 751
43      assign VS = ~((V < 490) | (V > 491)); //good
44
45      assign G2 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
46      assign G1 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
47      assign G0 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
48      assign R2 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
49      assign R1 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
50      assign R0 = ActiveRegion & (RightEdge | LeftEdge | TopEdge | BottomEdge);
51
52
53  endmodule
54
```

level(3:0)

clk

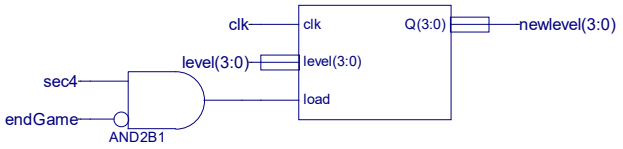
sec4

endGame

init

Slug Top Level

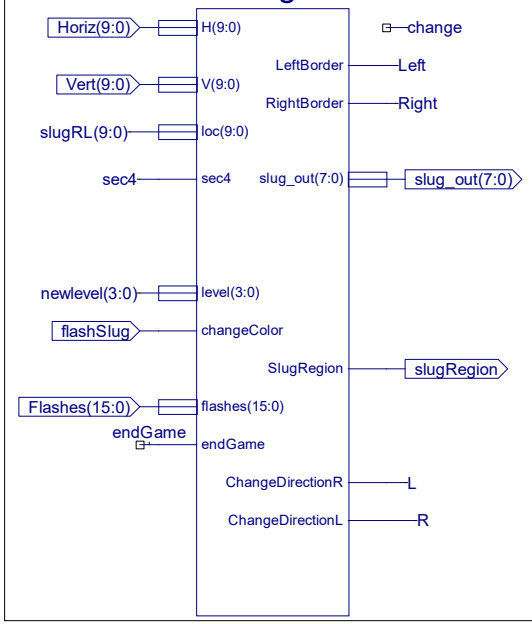
Register4bit



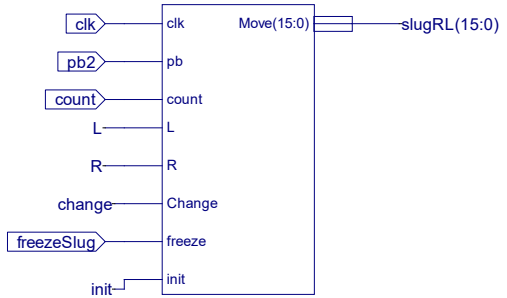
storing the level to calculate slug size

Verilog

Slug2



GetTurned2

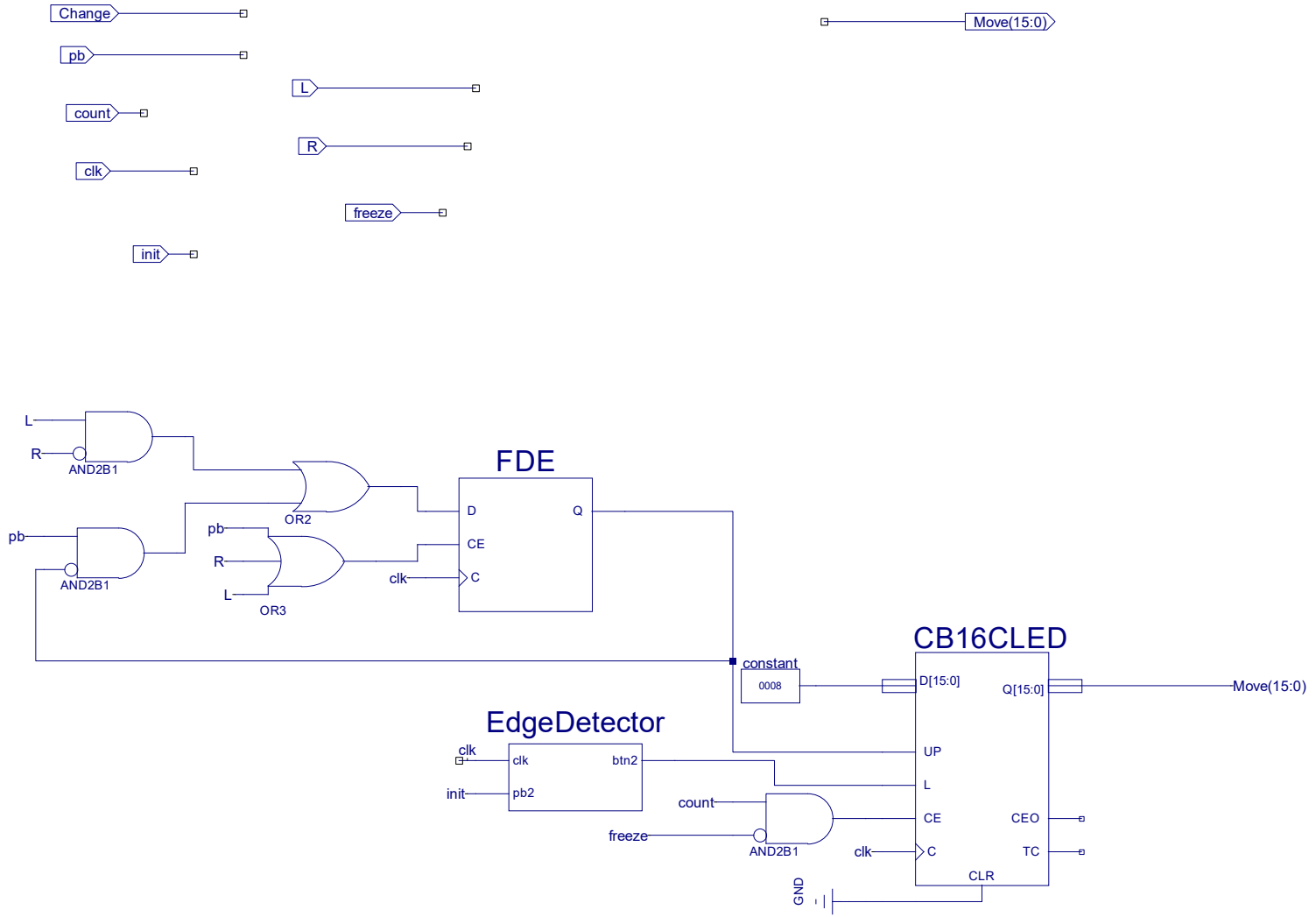


Turn direction and location logic

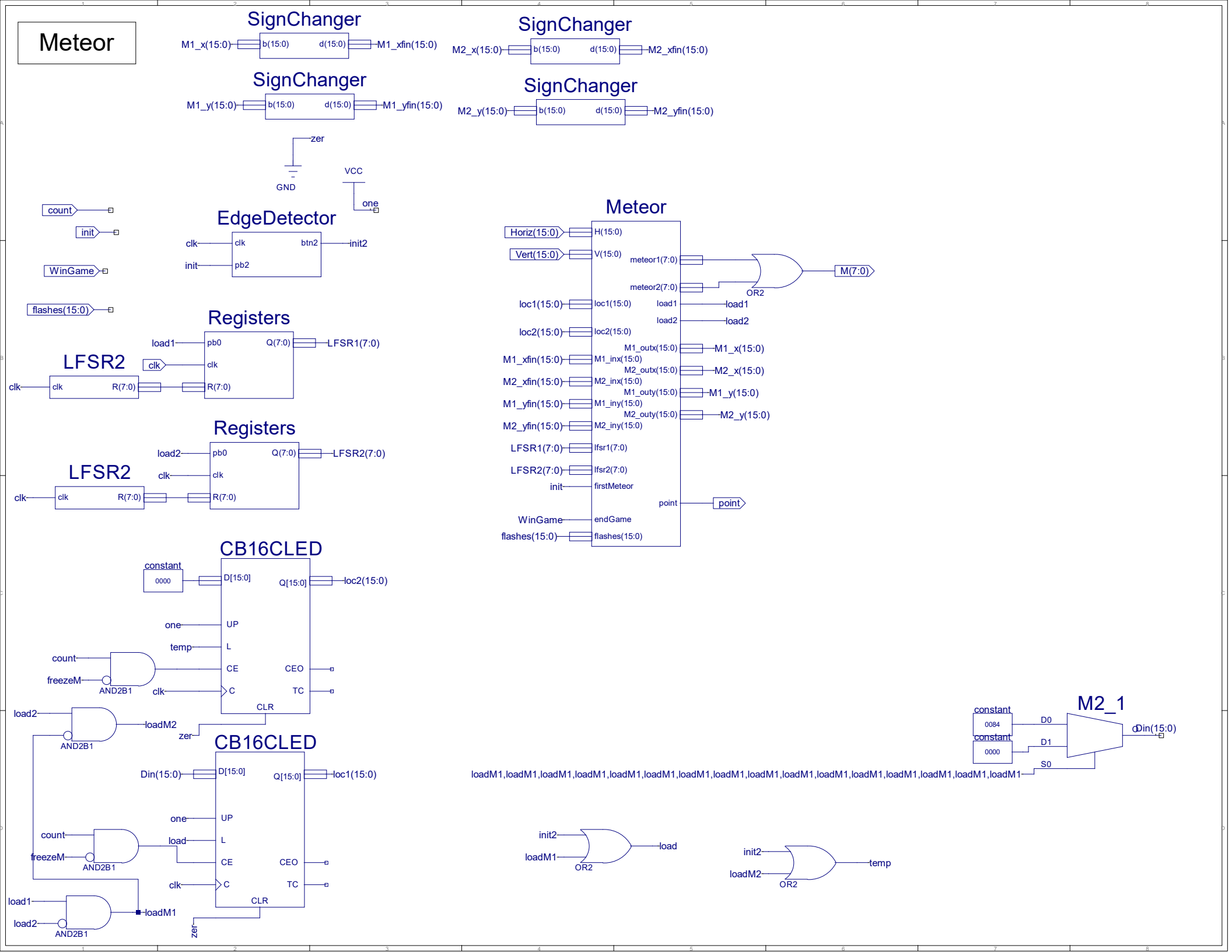
```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    14:35:28 05/16/2016
7   // Design Name:
8   // Module Name:    Slug2
9   // Project Name:
10  // Target Devices:
11  // Tool versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module Slug2(
22      input [9:0] H,
23      input [9:0] V,
24      input [9:0] loc,
25      input [3:0] level,
26      input changeColor,
27      input [15:0] flashes,
28      input sec4,
29      input endGame,
30      output [7:0] slug_out,
31      output ChangeDirectionR,
32      output ChangeDirectionL,
33      output LeftBorder,
34      output RightBorder,
35      output SlugRegion
36      );
37  wire RofSlug, LofSlug;
38
39  assign LeftBorder = (H<=8 & H>=0);
40  assign RightBorder = (H>=632 & H<=639);
41
42  assign RofSlug = 72 + (level*8) + loc;
43  assign LofSlug = 8- (level*8) + loc;
44
45  assign SlugRegion = (V <= 420 & V >= 404) & (H <= (72 + (level*8) + loc) & H >= (8- (
    level*8) + loc));
46
47  assign ChangeDirectionR =  SlugRegion&LeftBorder;
48  assign ChangeDirectionL =  SlugRegion&RightBorder;
49
50  //assign current_size = size;
51  assign slug_out [1] = (SlugRegion&~changeColor | SlugRegion&changeColor&flashes[5])&~
    endGame;
52  assign slug_out [0] = (SlugRegion&~changeColor | SlugRegion&changeColor&flashes[5])&~
    endGame;
53  assign slug_out [2] = SlugRegion&changeColor&flashes[5]&~sec4&~endGame | SlugRegion&
    endGame&flashes[4];
54  assign slug_out [3] = SlugRegion&changeColor&flashes[5]&~sec4 | SlugRegion&endGame&
```

```
      flashes[4];
55    assign slug_out [4] = SlugRegion&changeColor&flashes[5]&~sec4 | SlugRegion&endGame&
      flashes[4];
56    assign slug_out [5] = SlugRegion&endGame&flashes[4];
57    assign slug_out [6] = SlugRegion&endGame&flashes[4];
58    assign slug_out [7] = SlugRegion&endGame&flashes[4];
59
60    endmodule
61
```

Turn direction and Slug location logic



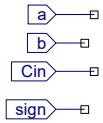
Meteor

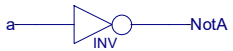

SignChanger
M1_x(15:0)
b(15:0)
d(15:0)
M1_xfin(15:0)
SignChanger
M2_x(15:0)
b(15:0)
d(15:0)
M2_xfin(15:0)
SignChanger
M1_y(15:0)
b(15:0)
d(15:0)
M1_yfin(15:0)
SignChanger
M2_y(15:0)
b(15:0)
d(15:0)
M2_yfin(15:0)
zer
GND
VCC
one
count
init
WinGame
flashes(15:0)
EdgeDetector
clk
clk
btn2
init2
init
pb2
Meteor
Horiz(15:0)
H(15:0)
Vert(15:0)
V(15:0)
meteor1(7:0)
meteor2(7:0)
OR2
M(7:0)
loc1(15:0)
loc1(15:0)
load1
load1
load2
load2
loc2(15:0)
loc2(15:0)
M1_outx(15:0)
M1_x(15:0)
M1_xfin(15:0)
M1_inx(15:0)
M2_outx(15:0)
M2_x(15:0)
M2_xfin(15:0)
M2_inx(15:0)
M1_outy(15:0)
M1_y(15:0)
M1_yfin(15:0)
M1_iny(15:0)
M2_outy(15:0)
M2_y(15:0)
M2_yfin(15:0)
M2_iny(15:0)
LFSR1(7:0)
lfsr1(7:0)
LFSR2(7:0)
lfsr2(7:0)
init
firstMeteor
point
point
WinGame
endGame
flashes(15:0)
flashes(15:0)
Registers
load1
pb0
Q(7:0)
LFSR1(7:0)
LFSR2
clk
clk
clk
clk
R(7:0)
R(7:0)
Registers
load2
pb0
Q(7:0)
LFSR2(7:0)
LFSR2
clk
clk
clk
clk
R(7:0)
R(7:0)
CB16CLED
constant
0000
D[15:0]
Q[15:0]
loc2(15:0)
one
UP
temp
L
count
CE
CEO
freezeM
AND2B1
clk
C
TC
CLR
load2
loadM2
AND2B1
zer
CB16CLED
Din(15:0)
D[15:0]
Q[15:0]
loc1(15:0)
one
UP
count
load
L
freezeM
CE
CEO
AND2B1
clk
C
TC
CLR
load1
loadM1
load2
AND2B1
zer
M2_1
constant
0084
D0
Din(15:0)
constant
0000
D1
S0
loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1,loadM1
init2
loadM1
OR2
load
init2
loadM2
OR2
temp

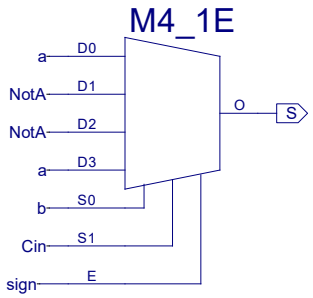
```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:01:18 05/15/2016
// Design Name:
// Module Name:    Meteor
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module Meteor(
   input [7:0] lfsr1,
   input [15:0] loc1,
   input [15:0] H,
   input [15:0] V,
   input [15:0] loc2,
   input [15:0] M1_inx,
   input [15:0] M2_inx,
   input [15:0] M1_iny,
   input [15:0] M2_iny,
   input [7:0] lfsr2,
   input firstMeteor,
   input [15:0] flashes,
   input endGame,
   output [7:0] meteor1,
   output [7:0] meteor2,
   output load1,
   output load2,
   output [15:0] M1_outx,
   output [15:0] M2_outx,
   output [15:0] M1_outy,
   output [15:0] M2_outy,
   output point
    );
    wire [15:0] center = 200;
   // wire meteor_1 =  (H <= (lfsr+32)) & (H >= (lfsr-32)) & (V > 0 +(loc1*2) & V <= 0+64+(loc1*2));// 200 = center
   // wire meteor_2 =  (H <= (lfsr+32)) & (H >= (lfsr-32)) & (V > 0 +(loc2 * 2) & V <= 0+64+(loc2*2));
    assign M1_outx = 264+lfsr1 - H;
    assign M2_outx = (lfsr2+80 - H)&~firstMeteor;
    assign M1_outy = (-40 - V + (loc1*2))&~firstMeteor | ((600 - V)+loc1*2)&firstMeteor ;
   assign M2_outy = (-90 - V  + (loc2*2))&~firstMeteor;

     //the formula:
      wire meteor_1 = (M1_inx  + M1_iny < 64) ;
      wire meteor_12 = (M1_inx +10 + M1_iny + 10) <64;
```

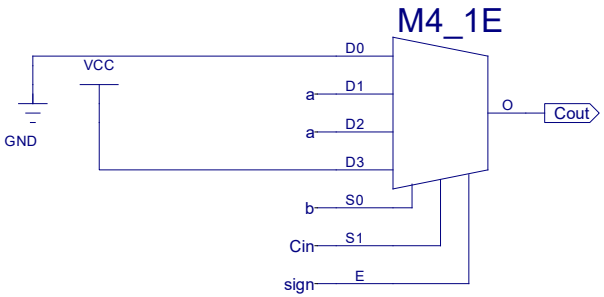
```
56          wire meteor_13 = (M1_inx + 20 + M1_iny + 20) < 64;
57          wire meteor_2 = (M2_inx  + M2_iny < 64);
58          wire meteor_22 = (M2_inx + 10 + M2_iny + 10) < 64;
59          wire meteor_23 = (M2_inx + 20 + M2_iny + 20) < 64;
60       // wire [15:0] maskx = tempx >> 15;
61       // wire [15:0] finalx = (maskx ^ tempx) - maskx;
62
63          //wire [15:0] tempy = center - V;
64          //wire [15:0] masky = tempy >> 15;
65          //wire [15:0] y = (masky ^ tempy) - masky;
66
67         // wire meteor_1 = y + finalx < 64;
68
69        //assign meteor[2] = meteor_0;
70        //assign meteor[3] = meteor_0;
71        //assign meteor[4] = meteor_0;
72
73    //bellow meteor is green
74        assign meteor1[7] = meteor_1&~endGame;
75        assign meteor1[6] = meteor_12&~endGame;
76        assign meteor1[5] = meteor_13&~endGame;
77        assign meteor1[4] = meteor_12&endGame&flashes[4];
78        assign meteor1[3] = meteor_1&endGame&flashes[3];
79        assign meteor1[2] = meteor_13&endGame&flashes[4] |  meteor_1&endGame&flashes[3];
80        assign meteor1[1] = meteor_12&endGame&~flashes[4];
81        // bellow meteor is red
82        assign meteor2[2] = meteor_23;
83        assign meteor2[3] = meteor_22;
84         assign meteor2[4] = meteor_2;
85         assign meteor2[7] = meteor_23;
86
87         assign load1 = (loc2 == 200);// was at 200
88         assign load2 =  (loc1 == 150); // was 150
89       assign point = (loc1 >= 280) | (loc2 >= 320); //was 280 and 320
90    endmodule
91
```

Adder

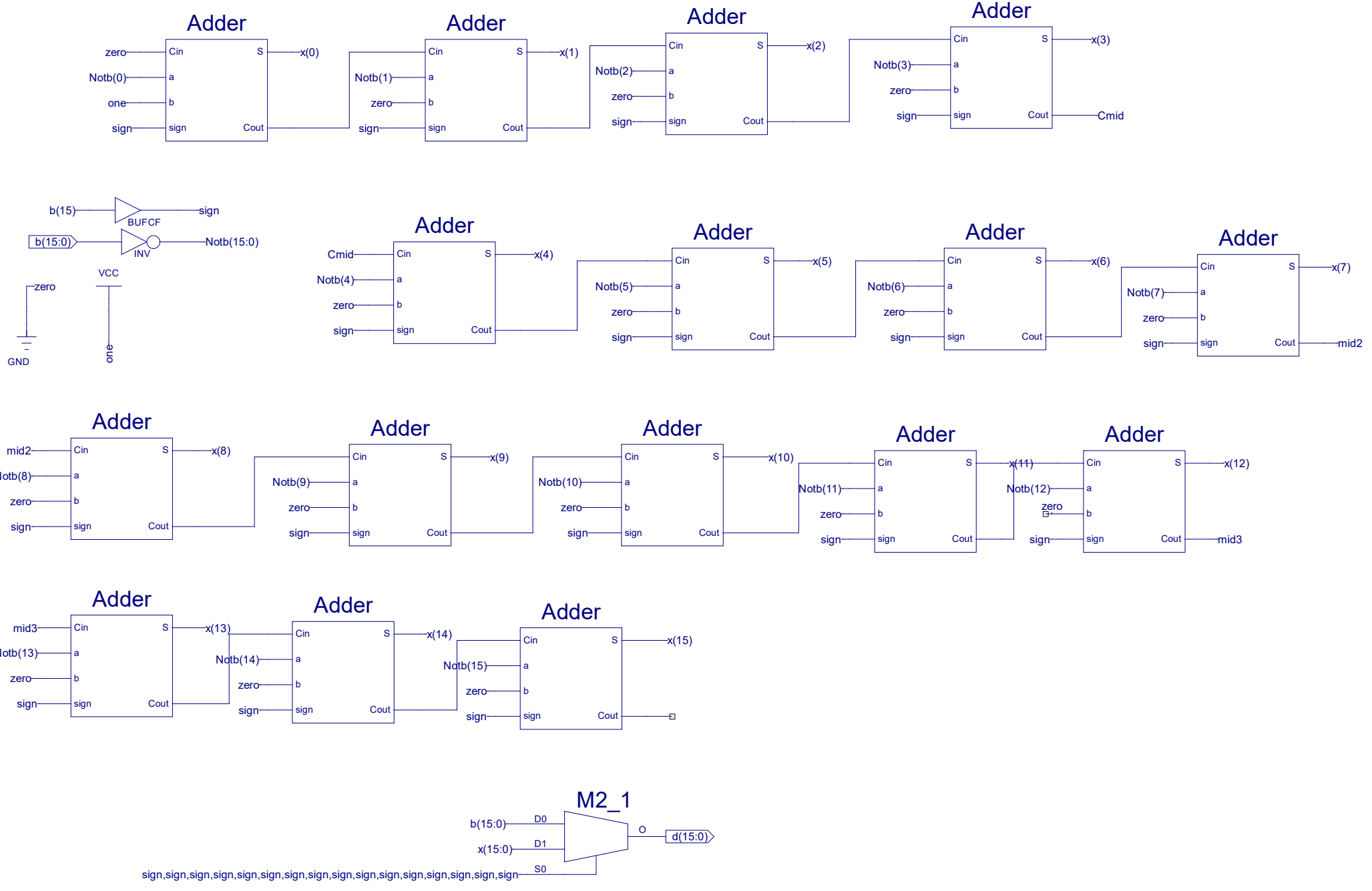

a
b
Cin
sign


a
NotA
INV

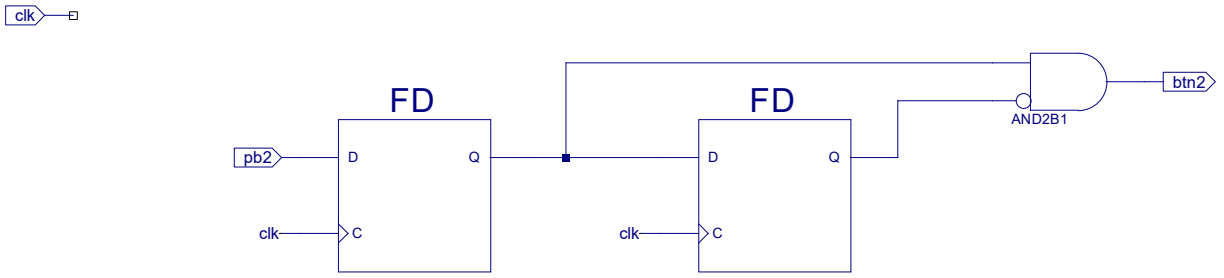

M4_1E
a
NotA
NotA
a
b
Cin
sign
D0
D1
D2
D3
S0
S1
E
O
S


M4_1E
VCC
GND
a
a
b
Cin
sign
D0
D1
D2
D3
S0
S1
E
O
Cout

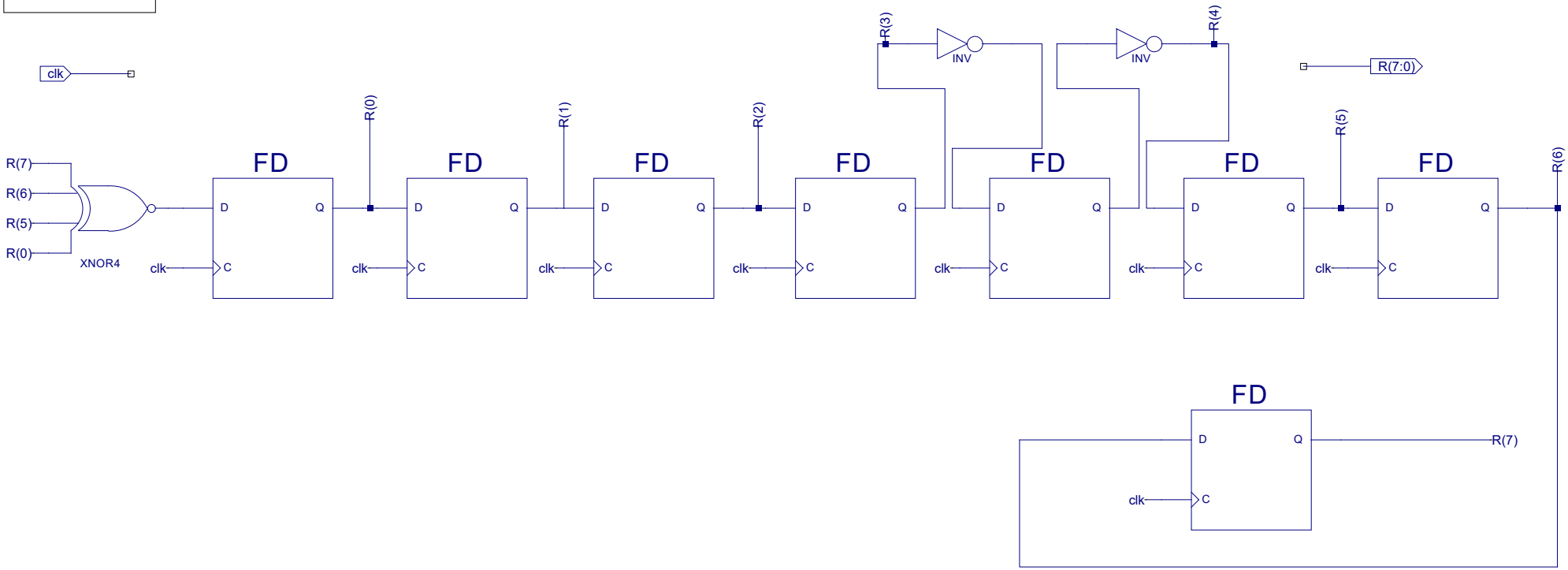
Sign Changer (2's Complement Calculator)



Edge Detection

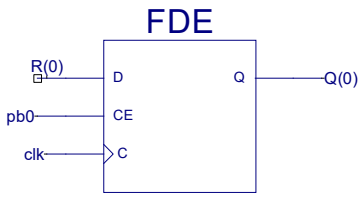


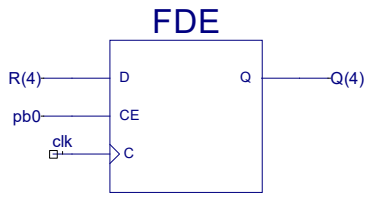
LFSR

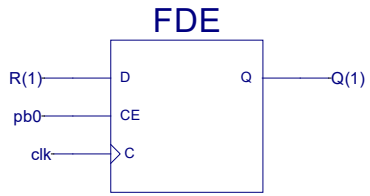


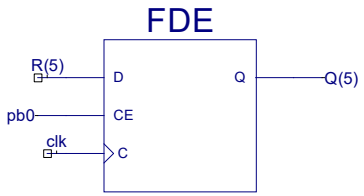
Registers

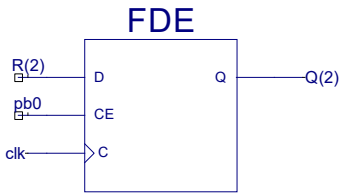

pb0
clk
R(7:0)
Q(7:0)

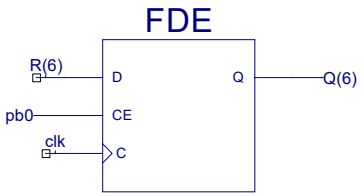

FDE
R(0)
D
Q
Q(0)
pb0
CE
clk
C

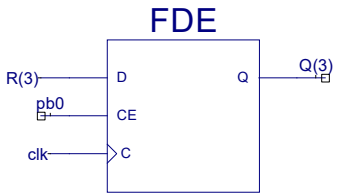

FDE
R(4)
D
Q
Q(4)
pb0
CE
clk
C

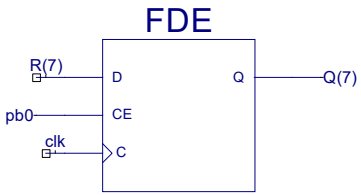

FDE
R(1)
D
Q
Q(1)
pb0
CE
clk
C


FDE
R(5)
D
Q
Q(5)
pb0
CE
clk
C

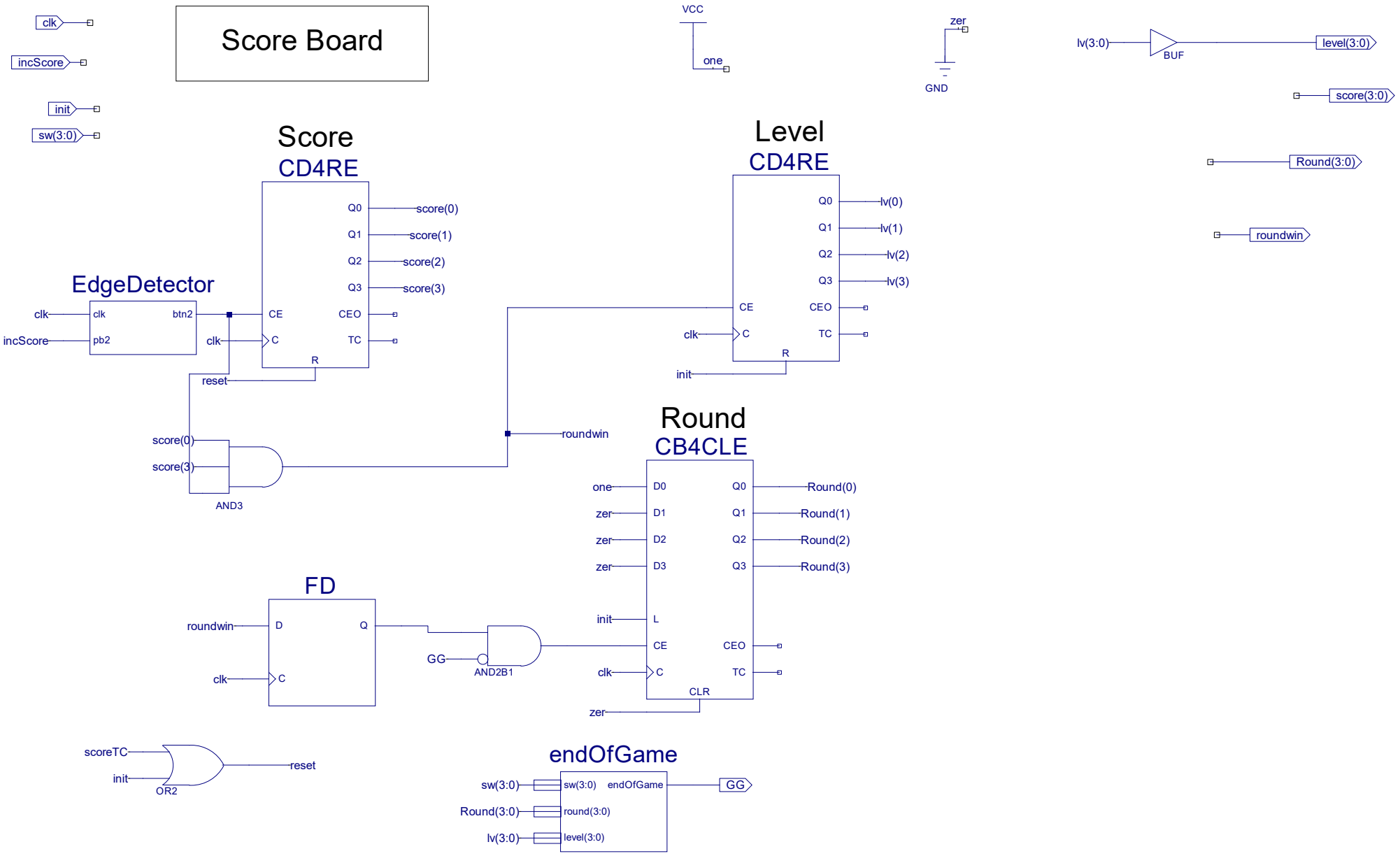

FDE
R(2)
D
Q
Q(2)
pb0
CE
clk
C


FDE
R(6)
D
Q
Q(6)
pb0
CE
clk
C

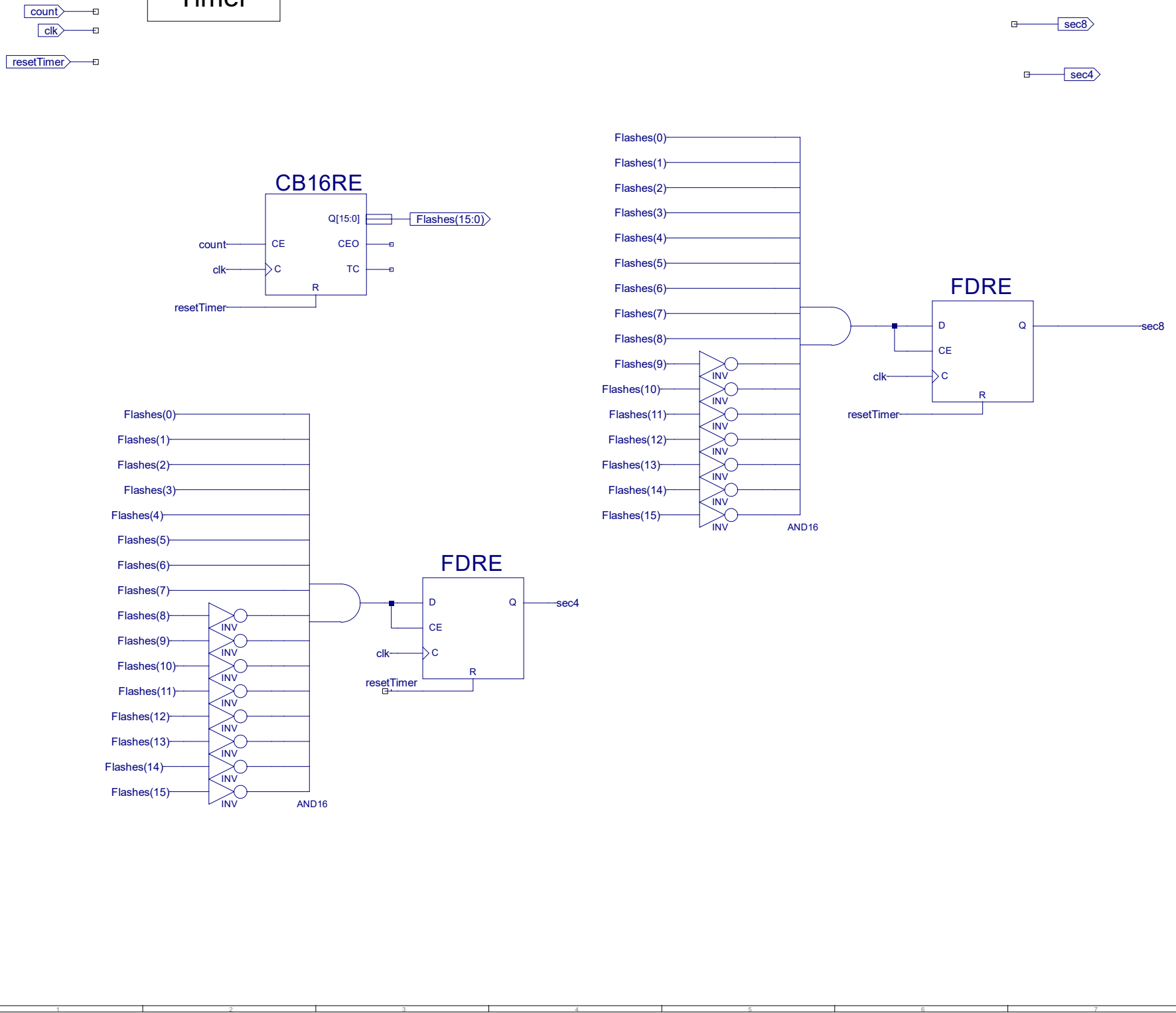

FDE
R(3)
D
Q
Q(3)
pb0
CE
clk
C


FDE
R(7)
D
Q
Q(7)
pb0
CE
clk
C

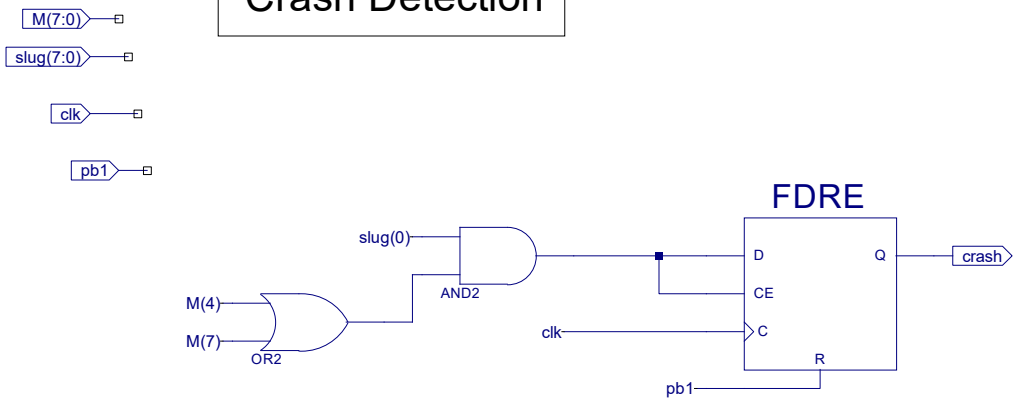
```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    16:34:31 05/23/2016
7   // Design Name:
8   // Module Name:    endOfGame
9   // Project Name:
10  // Target Devices:
11  // Tool versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module endOfGame(
22  input [3:0] sw,
23  input [3:0] round,
24  input [3:0] level,
25
26  output endOfGame
27      );
28
29  assign endOfGame = (sw == round) & (level == sw);
30
31  endmodule
32
```


Score Board
clk
incScore
init
sw(3:0)
VCC
one
zer
GND
lv(3:0)
BUF
level(3:0)
score(3:0)
Round(3:0)
roundwin
Score
CD4RE
Q0
Q1
Q2
Q3
score(0)
score(1)
score(2)
score(3)
CE
CEO
C
TC
R
EdgeDetector
clk
pb2
btn2
reset
Level
CD4RE
lv(0)
lv(1)
lv(2)
lv(3)
init
score(0)
score(3)
AND3
roundwin
Round
CB4CLE
one
zer
D0
D1
D2
D3
Round(0)
Round(1)
Round(2)
Round(3)
L
CLR
FD
D
Q
GG
AND2B1
scoreTC
OR2
endOfGame
sw(3:0)
round(3:0)
lv(3:0)
level(3:0)
GG

Timer

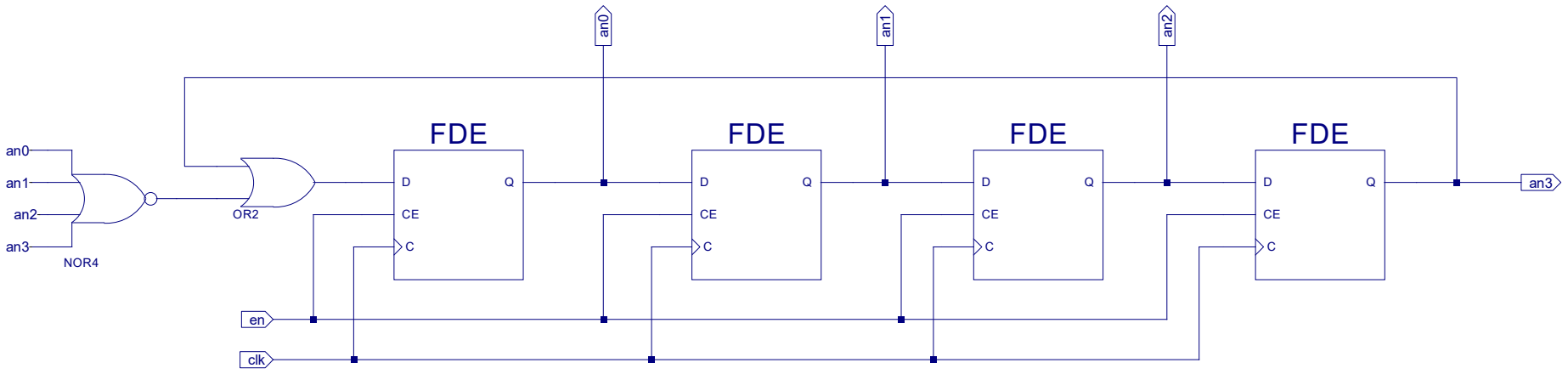

count
clk
resetTimer
sec8
sec4
CB16RE
Q[15:0]
Flashes(15:0)
CE
CEO
C
TC
R
Flashes(0)
Flashes(1)
Flashes(2)
Flashes(3)
Flashes(4)
Flashes(5)
Flashes(6)
Flashes(7)
Flashes(8)
Flashes(9)
Flashes(10)
Flashes(11)
Flashes(12)
Flashes(13)
Flashes(14)
Flashes(15)
INV
AND16
FDRE
D
Q

Crash Detection

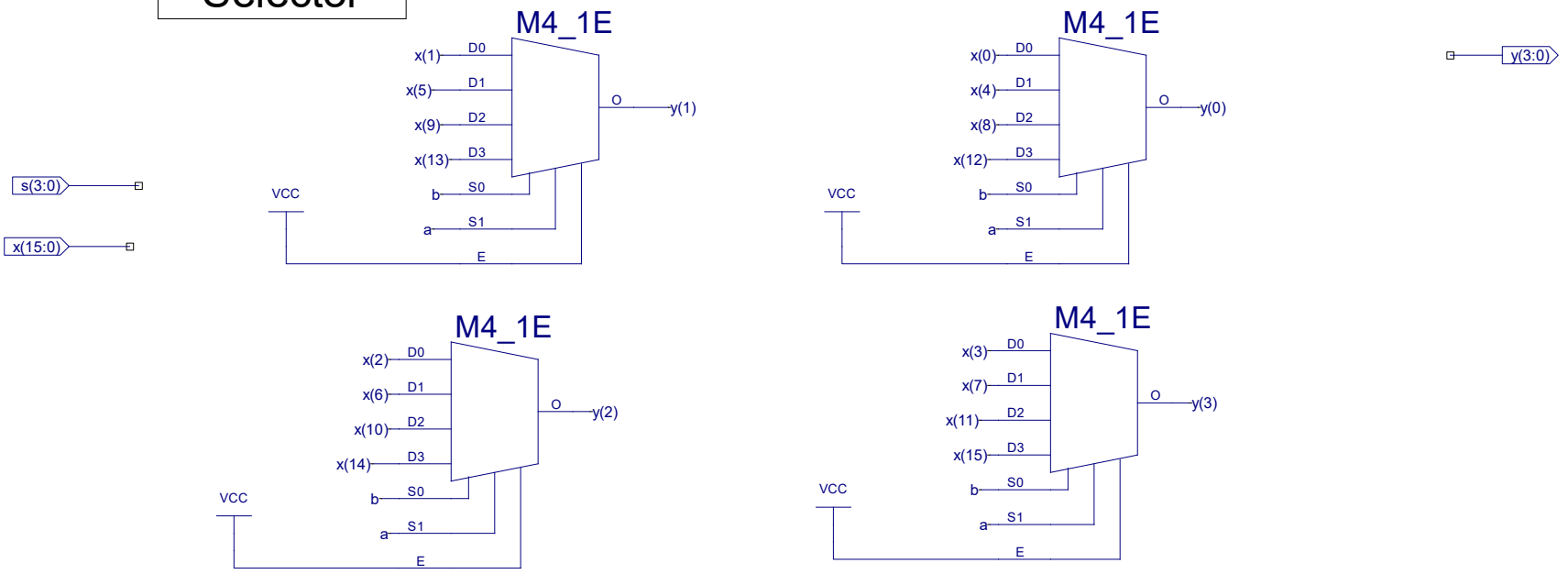


```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    19:09:35 05/14/2016
7   // Design Name:
8   // Module Name:    VGA_RBG
9   // Project Name:
10  // Target Devices:
11  // Tool versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module VGA_RBG(
22        input [7:0] RBG_values,
23        input [7:0] Slug_RBG,
24        input [7:0] Meteor_RBG,
25
26        output [7:0] RBG_out
27
28      );
29        assign RBG_out = RBG_values | Slug_RBG | Meteor_RBG;
30
31  endmodule
32
```

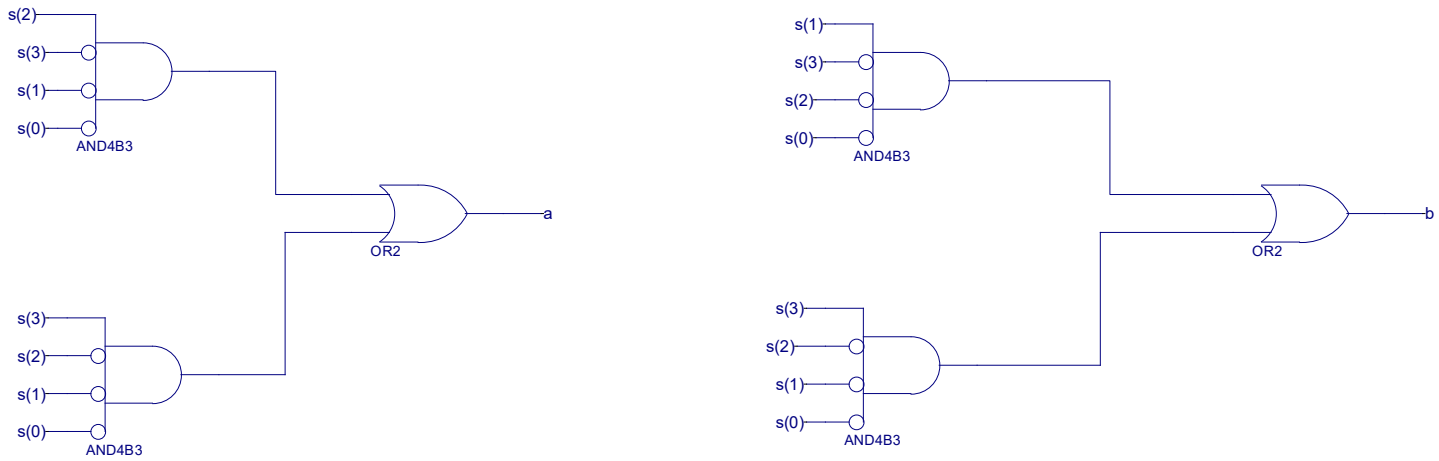
Ring Counter



Selector

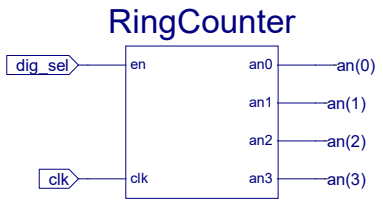


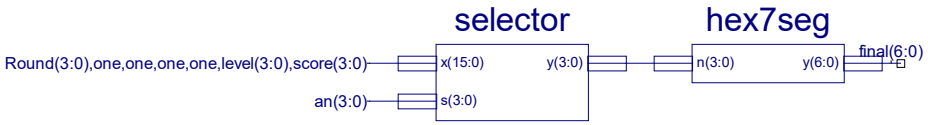
Selectors for muxes logic:

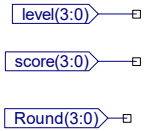


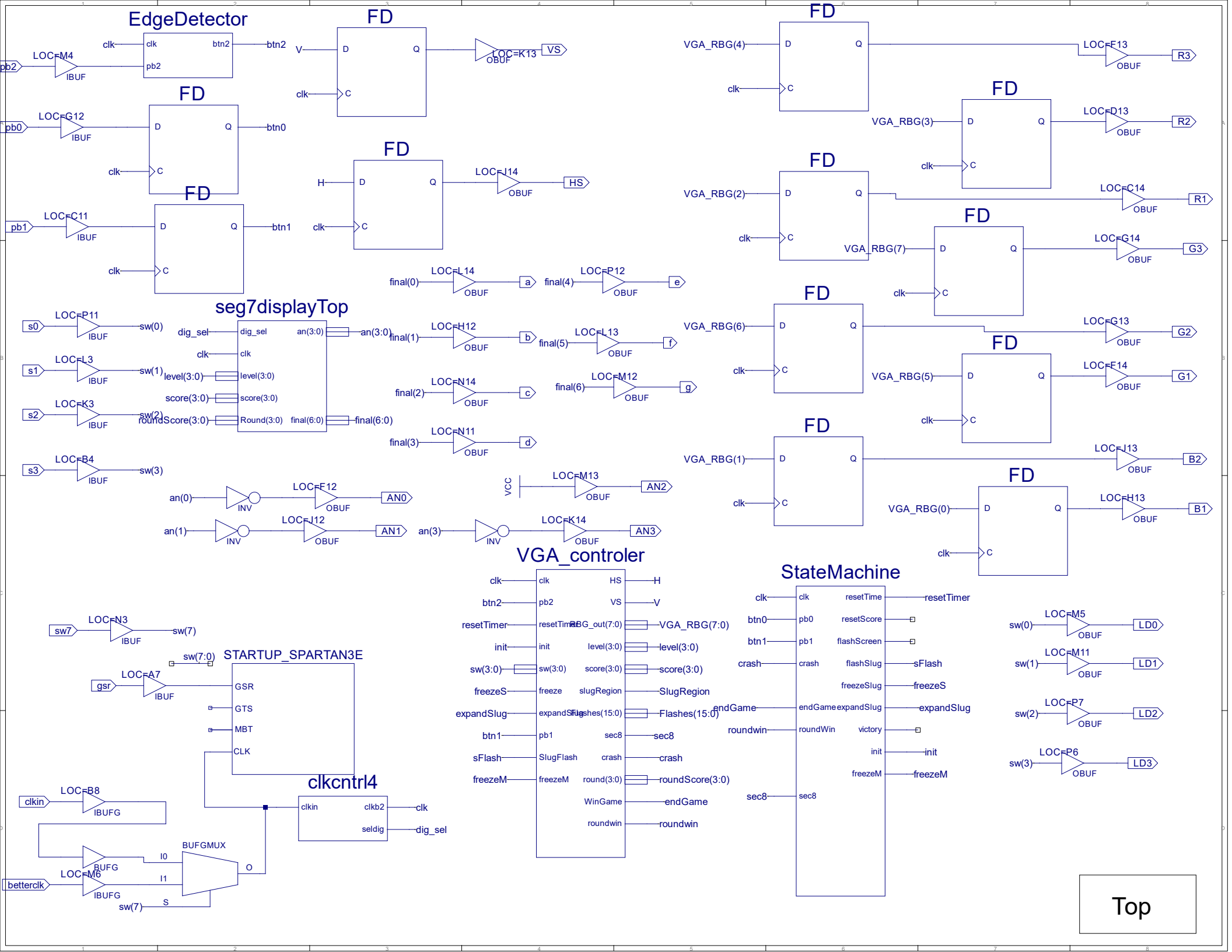
```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    19:55:37 04/30/2016
7   // Design Name:
8   // Module Name:    hex7seg
9   // Project Name:
10  // Target Devices:
11  // Tool versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21  module hex7seg(
22      input [3:0] n,
23      output [6:0]y
24      );
25
26     assign y[0] = (~n[3]&~n[2]&~n[1]&n[0]) | (~n[3]&n[2]&~n[1]&~n[0]) | (n[3]&~n[2]&n[1
    ]&n[0]) | (n[3]&n[2]&~n[1]&n[0]);
27     assign y[1] = (~n[3]&n[2]&~n[1]&n[0])  | (~n[3]&n[2]&n[1]&~n[0])  | (n[3]&~n[2]&n[1
    ]&n[0])  | (n[3]&n[2]&~n[1]&~n[0]) | (n[3]&n[2]&n[1]&~n[0])  | (n[3]&n[2]&n[1]&n[0]);
28     assign y[2] = (~n[3]&~n[2]&n[1]&~n[0])  | (n[3]&n[2]&~n[1]&~n[0])  | (n[3]&n[2]&n[1
    ]&~n[0])  | (n[3]&n[2]&n[1]&n[0]);
29     assign y[3] = (~n[3]&~n[2]&~n[1]&n[0])  | (~n[3]&n[2]&~n[1]&~n[0])  | (~n[3]&n[2]&n[
    1]&n[0])  | (n[3]&~n[2]&~n[1]&n[0]) | (n[3]&~n[2]&n[1]&~n[0])  | (n[3]&n[2]&n[1]&n[0]);
30     assign y[4] = (~n[3]&~n[2]&~n[1]&n[0])  | (~n[3]&~n[2]&n[1]&n[0])  | (~n[3]&n[2]&~n[
    1]&~n[0])  | (~n[3]&n[2]&~n[1]&n[0])  | (~n[3]&n[2]&n[1]&n[0])  | (n[3]&~n[2]&~n[1]&n[0
    ]);
31     assign y[5] = (~n[3]&~n[2]&~n[1]&n[0])  | (~n[3]&~n[2]&n[1]&~n[0])  | (~n[3]&~n[2]&n
    [1]&n[0])  | (~n[3]&n[2]&n[1]&n[0])  | (n[3]&n[2]&~n[1]&n[0]);
32     assign y[6] = (~n[3]&~n[2]&~n[1]&~n[0])  | (~n[3]&~n[2]&~n[1]&n[0])  | (~n[3]&n[2]&n
    [1]&n[0])  | (n[3]&n[2]&~n[1]&~n[0]);
33
34  endmodule
35
```

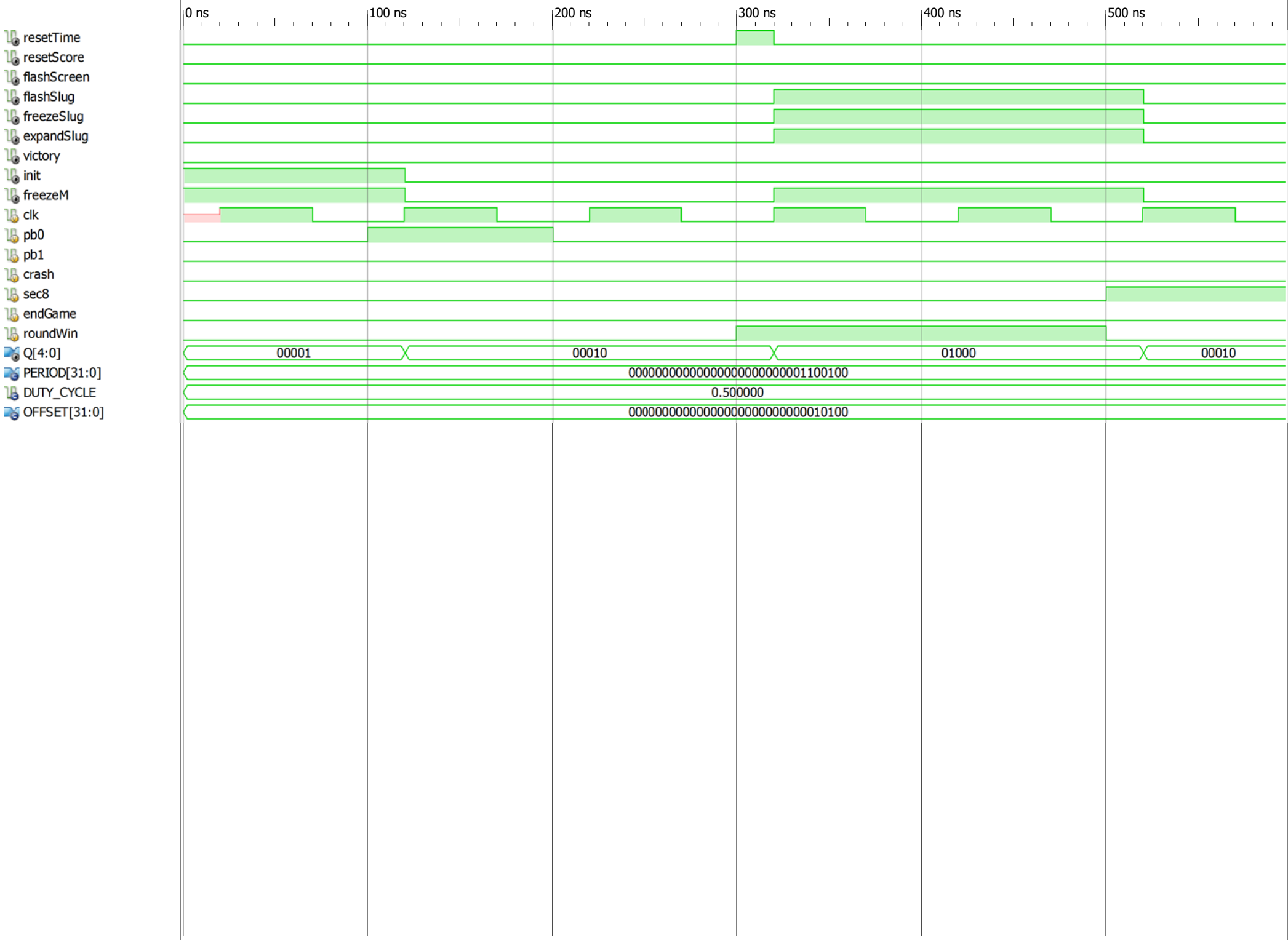
Display Logic








EdgeDetector
clk
btn2
pb2
LOC=M4
IBUF
pb2
FD
V
D
Q
C
clk
LOC=K13
OBUF
VS
FD
LOC=G12
pb0
IBUF
btn0
FD
LOC=C11
pb1
IBUF
btn1
FD
H
LOC=J14
HS
final(0)
LOC=L14
a
final(4)
LOC=P12
e
seg7displayTop
LOC=P11
s0
sw(0)
LOC=L3
s1
sw(1)
LOC=K3
s2
sw(2)
LOC=B4
s3
sw(3)
dig_sel
an(3:0)
level(3:0)
score(3:0)
roundScore(3:0)
Round(3:0)
final(6:0)
final(1)
LOC=H12
b
final(5)
LOC=L13
f
final(2)
LOC=N14
c
final(6)
LOC=M12
g
final(3)
LOC=N11
d
an(0)
INV
LOC=F12
AN0
an(1)
LOC=J12
AN1
VCC
LOC=M13
AN2
an(3)
LOC=K14
AN3
VGA_controler
clk
btn2
resetTimer
init
sw(3:0)
freezeS
expandSlug
btn1
sFlash
freezeM
HS
VS
RBG_out(7:0)
level(3:0)
score(3:0)
slugRegion
Flashes(15:0)
sec8
crash
round(3:0)
WinGame
roundwin
H
V
VGA_RBG(7:0)
SlugRegion
endGame
StateMachine
pb0
pb1
endGame
roundWin
sec8
resetTime
resetScore
flashScreen
flashSlug
freezeSlug
expandSlug
victory
resetTimer
sFlash
freezeS
init
freezeM
FD
VGA_RBG(4)
LOC=F13
R3
VGA_RBG(3)
LOC=D13
R2
VGA_RBG(2)
LOC=C14
R1
VGA_RBG(7)
LOC=G14
G3
VGA_RBG(6)
LOC=G13
G2
VGA_RBG(5)
LOC=F14
G1
VGA_RBG(1)
LOC=J13
B2
VGA_RBG(0)
LOC=H13
B1
LOC=M5
sw(0)
LD0
LOC=M11
sw(1)
LD1
LOC=P7
sw(2)
LD2
LOC=P6
sw(3)
LD3
sw7
LOC=N3
sw(7)
sw(7:0)
STARTUP_SPARTAN3E
gsr
LOC=A7
GSR
GTS
MBT
CLK
clkcntrl4
clkin
clkb2
seldig
clk
dig_sel
LOC=B8
IBUFG
BUFGMUX
BUFG
I0
I1
O
S
betterclk
LOC=M6
Top


0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
resetTime
resetScore
flashScreen
flashSlug
freezeSlug
expandSlug
victory
init
freezeM
clk
pb0
pb1
crash
sec8
endGame
roundWin
Q[4:0]
00001
00010
01000
00010
PERIOD[31:0]
00000000000000000000000001100100
DUTY_CYCLE
0.500000
OFFSET[31:0]
00000000000000000000000000010100

```
--------------------------------------------------------------------------------
Release 14.7 Trace  (nt64)
Copyright (c) 1995-2013 Xilinx, Inc.  All rights reserved.

C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\unwrapped\trce.exe -filter
C:/Users/rgnet/Desktop/final/iseconfig/filter.filter -intstyle ise -v 3
3 -fastpaths -xml Top.twx Top.ncd -o Top.twr Top.pcf

Design file:              Top.ncd
Physical constraint file: Top.pcf
Device,package,speed:     xc3s100e,cp132,-5 (PRODUCTION 1.27 2013-10-13
Report level:             verbose report

Environment Variable      Effect
--------------------      ------
NONE                      No environment variables were set
--------------------------------------------------------------------------------

INFO:Timing:2698 - No timing constraints found, doing default enumerati

INFO:Timing:3412 - To improve timing, see the Timing Closure User Guide
INFO:Timing:2752 - To get complete path coverage, use the unconstrained
   option. All paths that are not constrained will be reported in the
   unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are b
   a 50 Ohm transmission line loading model.  For the details of this m
   and for more information on accounting for different loading conditi
   please see the device datasheet.
INFO:Timing:3390 - This architecture does not support a default System
   value, please add SYSTEM_JITTER constraint to the UCF to modify the
   Uncertainty calculation.
INFO:Timing:3389 - This architecture does not support 'Discrete Jitter'
   'Phase Error' calculations, these terms will be zero in the Clock
   Uncertainty calculation.  Please make appropriate modification to
   SYSTEM_JITTER to account for the unsupported Discrete Jitter and Pha
   Error.



Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Setup/Hold to clock betterclk
------------+------------+------------+------------------+--------+
            |Max Setup to|Max Hold to |                  | Clock  |
Source      | clk (edge) | clk (edge) |Internal Clock(s) | Phase  |
------------+------------+------------+------------------+--------+
pb0         |    3.108(R)|   -1.519(R)|clk               |   0.000|
pb1         |    3.437(R)|   -1.782(R)|clk               |   0.000|
pb2         |    2.645(R)|   -1.148(R)|clk               |   0.000|
```

s0	10.294(R)	-3.453(R)	clk	0.000
s1	11.422(R)	-4.353(R)	clk	0.000
s2	10.557(R)	-3.662(R)	clk	0.000
s3	10.768(R)	-3.831(R)	clk	0.000

Clock betterclk to Pad

Destination	clk (edge) to PAD	Internal Clock(s)	Clock Phase
AN0	5.326(R)	clk	0.000
AN1	4.866(R)	clk	0.000
AN3	4.714(R)	clk	0.000
B1	4.658(R)	clk	0.000
B2	4.835(R)	clk	0.000
G1	4.377(R)	clk	0.000
G2	4.662(R)	clk	0.000
G3	5.139(R)	clk	0.000
HS	5.138(R)	clk	0.000
R1	4.386(R)	clk	0.000
R2	4.861(R)	clk	0.000
R3	4.657(R)	clk	0.000
VS	4.634(R)	clk	0.000
a	10.422(R)	clk	0.000
b	10.790(R)	clk	0.000
c	9.859(R)	clk	0.000
d	9.909(R)	clk	0.000
e	10.177(R)	clk	0.000
f	10.219(R)	clk	0.000
g	9.576(R)	clk	0.000

Clock to Setup on destination clock betterclk

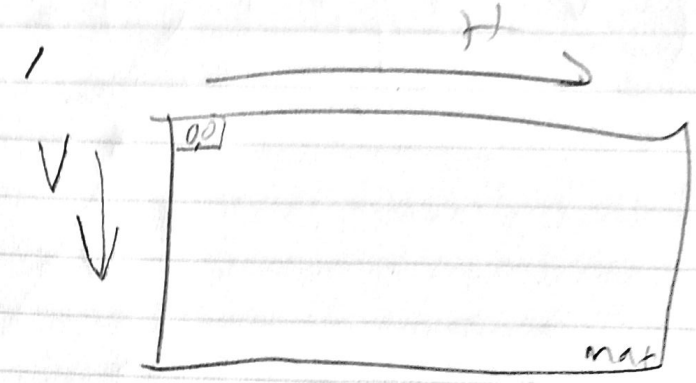
Source Clock	Src:Rise Dest:Rise	Src:Fall Dest:Rise	Src:Rise Dest:Fall	Src:Fall Dest:Fall
betterclk	30.503			

Pad to Pad

Source Pad	Destination Pad	Delay
s0	LD0	4.882
s1	LD1	5.962
s2	LD2	6.763
s3	LD3	5.954

```
Analysis completed Wed Jun 01 21:38:11 2016
--------------------------------------------------------------------------

Trace Settings:
-------------------------
Trace Settings

Peak Memory Usage: 175 MB
```

Early Bonus 1
Tu 5/17
3105




```
for (Vertical) {
    for (Horiz) {
        count++;
    }
}
```

→ 2 Counters

→ Horiz TC increments Vertical

Lab 7

Good job! 3105 5:38 pm 5/24/16