TIN THUREIN

LOGIC DESIGN: BALL CATCHING GAME WITH FPGA

FINAL PROJECT WRITE UP

COMPUTER ENGINEERING 100

LOGIC DESIGN

11 MARCH 2016

UNIVERSITY OF CALIFORNIA, SANTA CRUZ

Description:

For lab 7 we design a game that which "catches" the "ball" between the two vertical paddles. The ball consists of 8x8 pixels square block, bouncing on the screen, between the top of the "play region" and the carpet. The screen is of 640x480 pixel; the play region is 624x216 pixel. Basically, the player gets 3 paddles to play with, and every time the ball goes through the middle section of the paddle, he scores and he loses a paddle if the ball hits the paddle. The player also loses the paddle and scores if the ball goes through the middle section but hit the paddle afterward.

Methods:

This lab tested out all of our knowledge on designing with logic. Once I get the hang of getting the displaying the play region and such, it was not that hard anymore. The first thing we needed to do was to display the active region which is a simple blue background, of 640x480 pixels. And the play region which is 256 pixels down, and 8 pixels away from the border of the screen. We also needed to implement H-sync and V-sync to correctly sync with the monitor and be able to send data via VGA cable. The VGA connector consists of 8 color bits—3 red, 3 green, and 2 blue—and 2 output for H-sync and V-sync. We use counters to show individual pixels on the screen: one for horizontal and one for vertical. Just like we normally would write on paper, we "write" each pixel by counting up the horizontal first and once we reach to certain pixel number, we reset the counter and advance the vertical counter by one, in order to write the next line of pixel. To do so, we needed a pretty fast clock to trick our eyes that the pixels aren't written over and over again. For the moving parts in our game, we used counters to move each pixel. For the moving parts in our game, we used counters to move each pixel on individual object and a dedicated counter for each motion of the object. For example, we have to move the ball by way of bouncing; to do so, we use two dedicated counters that one counts either up or down, and one that always count up.

Results:

<u>The Controlling Logic:</u>

The most important part of this lab is controlling the positions of each pixels and when to turn on or off. I have written in Verilog to do so. All the Verilog code of this controlling logic can be seen in the figure 15 and 16.

--Carpet region:

I first defined the carpet region in Verilog, where the carpet is 16 pixels in height and has the length of 528 pixels.

--Paddle location:

I split the paddle region into x and y components and define where they belong initially. Since we won't be moving the paddle in y-direction we don't need a counter to move it; only two counters are needed to move it in x-direction. I first defined where its initial positions are, namely at 239 and 255. I then moved onto add one counter and subtract another counter to either move left or right. I then "ANDed" with the y component of the paddle to make sure it only turns on when the pixels are at the exact location of x and y.

--Ball location:

Same again with the ball as I split into two components. I first define where the ball initial location in y-direction, namely 256 and 263 (8 pixels long), and add it to the counter to bring it down.

For the x-direction of the ball, I define its initial position and subtract to move it to the left.

To start the ball at randomized location, I implemented by adding the number from the random number generator to its initial position in y-direction.

--Start the ball at random place:

To start the ball at random place after the ball has passed the left side of the play region is done by having a signal goes high when the x location of the ball passed a certain boundary, which is at the 7^{th} pixel on the screen. If this signal goes high, we know we have the ball passed and have the up down counter load the random number + the ball's initial position to get a random place to start.

--When the paddle dropped off the carpet:

I define a signal called "freeze" when the paddle reached over the carpet and fall. And I also made a signal which goes high when the paddle has fallen to the bottom of the play region. This signal tells me to reset the counters and begin the game if we still have the paddles.

--Collision with paddle and ball:

This signal goes high when we have the paddle location is the same as ball location in both x and y direction.

--Moving the carpet with two different colors:

I first defined an internal wire of 16 bits, which is the x location on the screen + the counter for the carpet. I then took the 5^{th} bit or the 32 pixels wide and "ANDed" with the carpet region in both x and y direction. For another color of the carpet, I defined the inverse of the 5^{th} bit, and do

the same. This gives us two section of 32 bits wide pixels which we can use it to connect different color outputs.

Defining regions (pixel generator):

To get the play region, active region, and H-sync and V-sync, we use two counters. Instead of using comparators to compare the pixel positions, I used Verilog code that defines each region on the screen. Please see figure 13 for implementation of this.

Moving the paddle:

For the paddle, I initially implemented the up-down counter to move it either left or right depending on the PB0 being pressed or not. I soon realized that I would need to convert the bits to 2's complement in order to move left. So instead I used two 16 bits counters to move either left or right. Basically I needed to make sure that only one of those counters was counting up or else we would have the paddles stationary. I used 5 inputs AND gate to make sure the paddle only moves every frame of the screen, and only when I don't have collision with the ball. Below is the implementation of this.

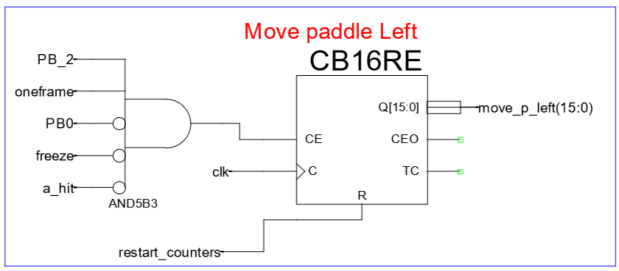

Figure 1. Implementation of moving the paddle to the left

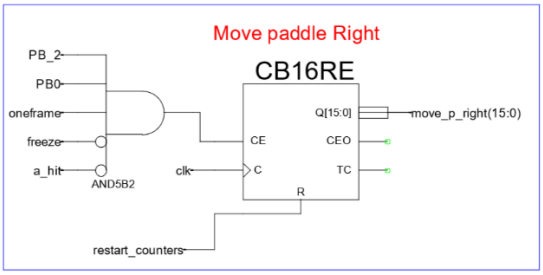

Figure 2. Implementation of moving the paddle to the right

The trick involves in the top-level implementation is that we need to somehow hold the value of the collision and PB2 until we get a collision or paddle drop. To do so, I needed to use a flip flop with the enable pin, as well as the clear pin and tie the enable pin along with the input of the flip flop. So once the signal goes high for a brief moment, we have the value capture for the entire time until we reset it.

Bouncing the ball:

We also need two counters for the ball to move. I simply used two counters to move left and move up and down. To move up and down, I used a simple up-down counter which the UP input pin of the counter of the pin is tied to the signal called "hit_ceiling", which basically is the signal that tells when the position of the ball has hit the top of the play region. When this signal goes high, we need to bring the ball back down. In order to do so, we need to "hold" its value until we hit the top of the carpet. To "hold" value I used FDCE, the flip flop with count enable bit and

the input D. I tied the signal “hit_ceiling” to the input D and the CE bit of the flip flop. This allows the flip flop to hold its value because when the CE bit goes low, the output of the flip flop doesn’t change. As we said earlier, we clear the value of this flip flop when we hit the top of the carpet, which I called this signal “hit_floor”.

To start the ball at random position, once the ball has passed the left side of the screen was done so by the same counter. It has “load” pin and we tied it to the signal called “far_left” which simply goes high when the ball has passed the left side bound of the play region. Once this signal goes high, we tied its load input value to the value from the random number that we generated.

Moving the ball to the left was quite simply. We only needed to use a 16 bits counter that counts up when we don’t get a hit and while we still have the paddle left to play with. Both of these counters get reset when we have no more paddle left, or when we get a hit or when the paddle has dropped from the carpet.

Counting up the score:

Defining the region to says count up the score was a little difficult. I first implemented that we count up the score if we have the ball location y = paddle middle section y and ball location x = paddle middle section x. For some weird reason, I was never able to get the signal to count up. So I had to implement another way, which was to define a small region on the ball, instead of defining the whole ball region, and when that small region is in the same location as the paddle middle section location, I have the signal to output 1. This works well in a sense that I could see my counter counting up but it also had a few glitches where it would count up to a multiple of 4 or so. I thought perhaps the signal was longer than a clock cycle and having an edge detector would help to count up only once no matter how long the signal was. Unfortunately, that also didn’t help. I thought perhaps my edge detector from previous lab was not working correctly, so I implemented another one on the top level to see if there was any change to be seen. That also didn't improve the situation.

So I implemented the region on the ball even smaller and the region on the paddle to be larger. So basically I defined a bottom left region of the ball, which is a pixel or two wide and two pixel height, and use that region to compare against the paddle region. It turns out it was the way to go as I was able to get the number to count up consecutively.

Moving the carpet:

Moving the carpet was simply as we only need it move left as long as we don’t get a collision and as long as we have the paddle to play with. It gets reset when we have 6^{th} bit of its signal goes high, or the 2^6 or 64 pixel has reached (because we take 32 pixels to be one color and the other 32 pixels to be another color).

Dropping off the carpet:

When we have the paddle moved passed the carpet’s edge, we have the signal called “freeze”, which freezes, as the name implies, all the motion of game, and let the paddle drops at 2

times the pixel per frame. We reset this counter when we have the signal “restart counters” goes high.

After the paddle dropped to the floor of the play region, the game restarts if the player still have the paddle to play with. Otherwise, the ball and the carpet continues moving, until the play pressed PB1 to restart the game.

Random number generator:

I reused the same random number generator from the previous lab. This time we need to feed it to the 16-bits bus, so we add the zeroes in front to make it a 16-bits number. This number gets fed to the up down counter of the ball. Also I only used the 5:0 bits as we only wish to have the ball starting at reasonable position and not at very low.

Flashing the paddle:

To flash the paddle, and pause everything on the screen, is done so by an 8-bits counter where its 5^{th} bit gets ANDed with the output of the paddle. What this means is that when we get a collision signal, we have the counter to count up every clock cycle and reset it when we have the upper 4 bits, bit 7,6,5,4 are high. We reset all the counters and begin the game again if we have the paddle left. This ultimately creates the flashing effect, as the 5^{th} bit goes high to low several times before it gets reset.

State machine:

The machine is only needed to have the ball and the carpet moving after losing all the paddles, and load 3 paddles when we begin playing. The State machine consists of only 3 states, the idle state, play state and the lost state. It only has 3 output signals: to load paddle, to move carpet and to move the ball.

One mistake I made during this lab was that I had a typo at the input Present State to [0:2] instead of [2:0]. I was not aware of this mistake and I found my typo when I run a simulation of the state machine. When I generated the bit file and uploaded to the FPGA board, it seemed as if the game always starts at a different state. I noticed that because the paddle is always 0 when the game starts and nothing happens when I pressed the PB2. But when I pressed PB1, it seemed the game begins normally. Even though all the wires and states are defined correctly, the typo has caused a huge headache.

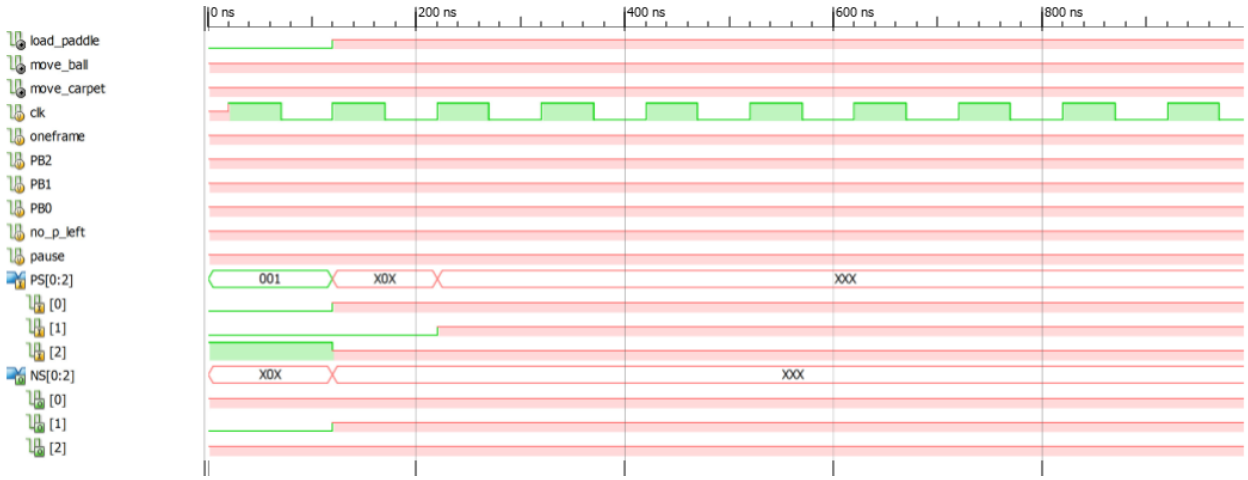

Figure 3: As we can see, we always have the state starting with the last state because of the typo.

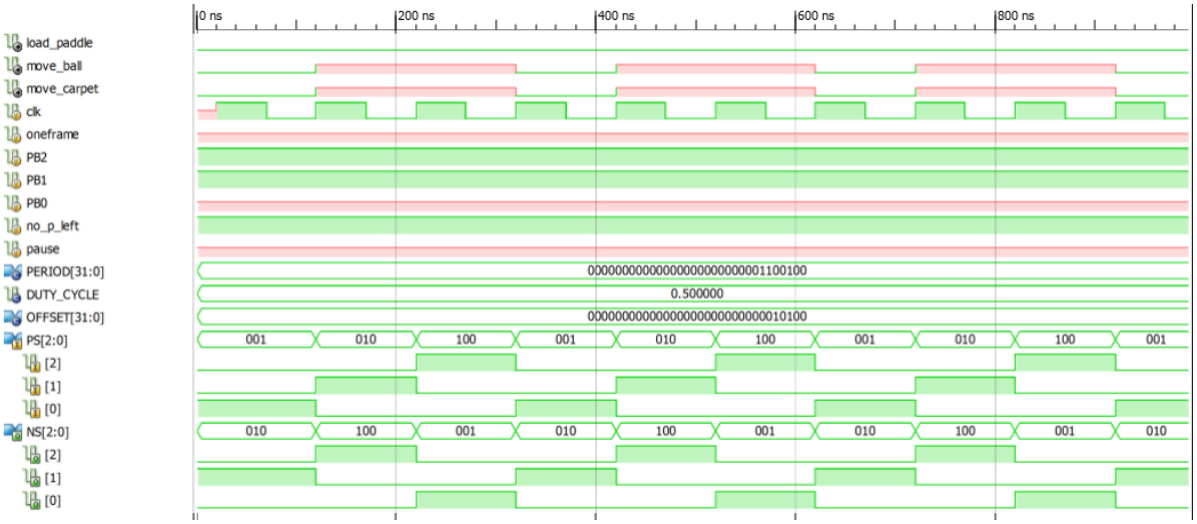

Figure 4: I simulated with a few variations to make sure the state machine is fully working.

The implementation of state machine can be seen in the figure 45.

Paddle lives:

I used a 2 bits up-down counter, which only counts down. I tied the load pin to the signal coming out from the state machine which tells the counter to load 3 paddles when the game begins. Whenever we get a hit or falls off of the carpet, we count down. We compare this value against 0 in the comparator to see if we have any more paddles left. When we don't have any more paddles, we output a signal that goes directly to the state machine and let the ball and carpet running without the paddle showing.

On the top level, all we need is to take the color bits (i.e. paddle, carpet, ball) and connect those to the RGB bits.

Conclusion:

Lab 7 was a very much challenging lab as we needed to figure things out using the prior knowledge. I was not able to get a computer station on the second week of the lab to test out my implementation so I was not sure if my program was going to work at all. I came to check several time but no one left and I couldn't wait for hours. So instead, I just tried to get as much as I could on my own and on Thursday, I came to my lab section and tested out my program. It turns out it was not working at all. Being frustrated, I couldn't debug for I was not sure which part/parts were not working correctly. It took me another day or two to get it working properly. If I could have done something differently, I would have done so by starting out early on the first week and get as much done by the weekend. And perhaps being carefully while writing in Verilog could have save me a lot of times.

Question:

From the edge of the clock to the input pin, the maximum time it takes is 7.022ns, which converts to about 142Mhz.

From the edge of the clock to output pin, the maximum time it takes is 14.655ns, which converts to 68Mhz.

Since I didn't use flip flops for the outputs, the maximum time is still the same.

Appendices:

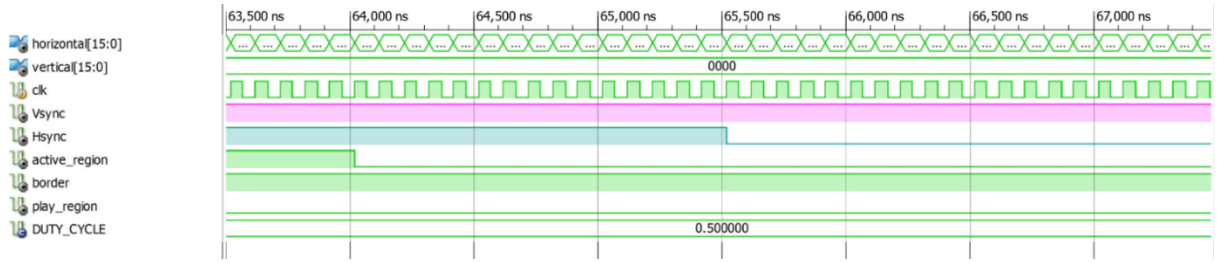

Figure 5: The H-sync signal goes low for a brief period.

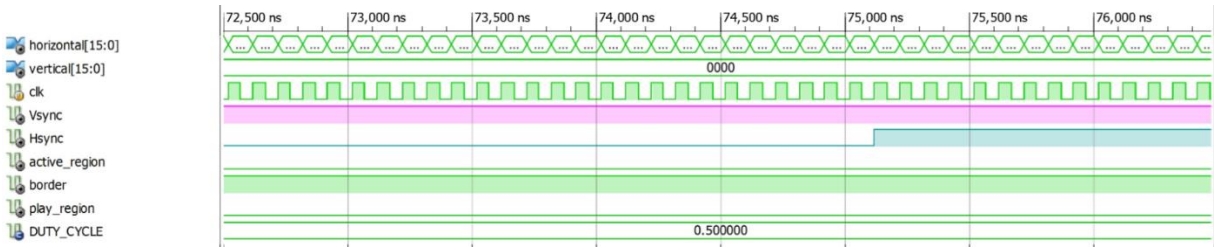

Figure 6: The H-sync goes high again. It takes about 9500ns (75000ns-65500ns) to go high again.

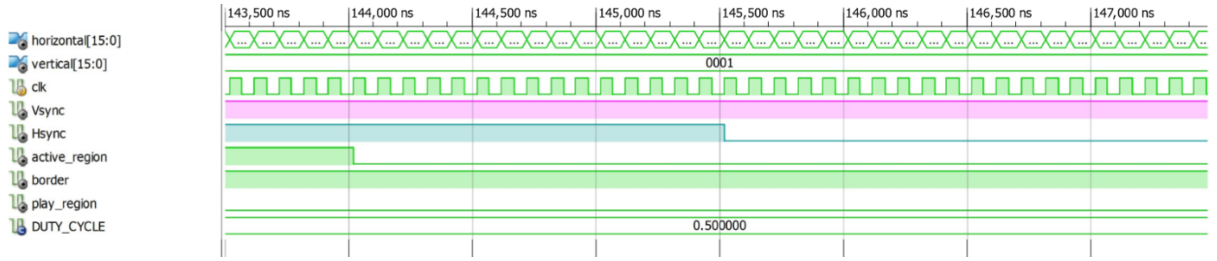

Figure 7: The H-sync goes low again at 145,500ns

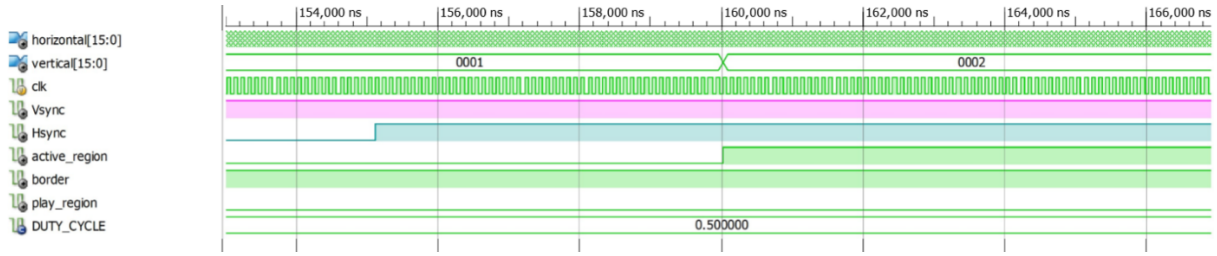

Figure 8: The H-sync goes high and it takes 9500ns. It seems as if the period that goes low is the same and the time that it goes high again is the multiple of 2xthe first time that it goes high again.

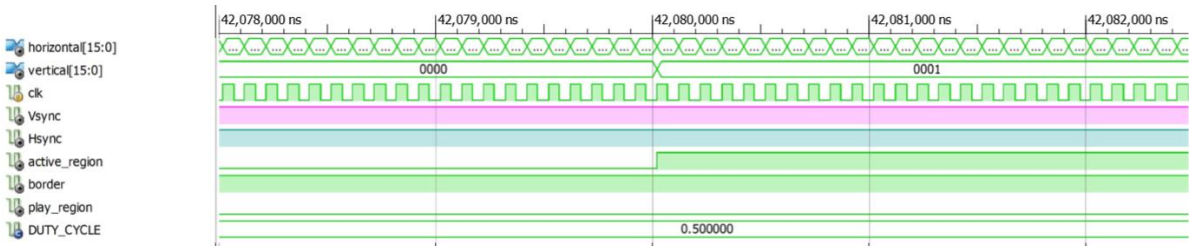

Figure 9: With the clock period of 100ns, it takes about 0.042 second to write one frame. That is the reason we needed a 25Mhz clock, or a 40ns clock. With 40ns, we could write about 60 frame in a second.

```
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  |
------------+------------+------------+------------------+--------+
P_B0        |    5.967(R)|   -2.878(R)|clk               |   0.000|
P_B1        |    6.657(R)|   -2.272(R)|clk               |   0.000|
P_B2        |    7.022(R)|   -2.207(R)|clk               |   0.000|
------------+------------+------------+------------------+--------+

Clock betterclk to Pad
------------+------------+------------------+--------+
            | clk (edge) |                  | Clock  |
Destination |   to PAD   |Internal Clock(s) | Phase  |
```

```
------------+------------+------------------+--------+
AA          |   10.164(R)|clk               |   0.000|
AB          |   10.980(R)|clk               |   0.000|
AC          |   10.324(R)|clk               |   0.000|
AD          |   10.374(R)|clk               |   0.000|
AE          |   10.403(R)|clk               |   0.000|
AF          |   10.352(R)|clk               |   0.000|
AG          |   10.237(R)|clk               |   0.000|
AN0         |    5.405(R)|clk               |   0.000|
AN1         |    4.746(R)|clk               |   0.000|
AN3         |    4.927(R)|clk               |   0.000|
B0          |   11.748(R)|clk               |   0.000|
B1          |   11.423(R)|clk               |   0.000|
G0          |   13.457(R)|clk               |   0.000|
G1          |   14.211(R)|clk               |   0.000|
G2          |   14.655(R)|clk               |   0.000|
HS          |    8.883(R)|clk               |   0.000|
Hsync       |    9.469(R)|clk               |   0.000|
R0          |   12.032(R)|clk               |   0.000|
R1          |   13.229(R)|clk               |   0.000|
R2          |   13.233(R)|clk               |   0.000|
VS          |   10.130(R)|clk               |   0.000|
Vsync       |    9.642(R)|clk               |   0.000|
------------+------------+------------------+--------+

Clock to Setup on destination clock betterclk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
betterclk      |   11.527|         |         |         |
---------------+---------+---------+---------+---------+
```

Figure 10: The Timing analyzer

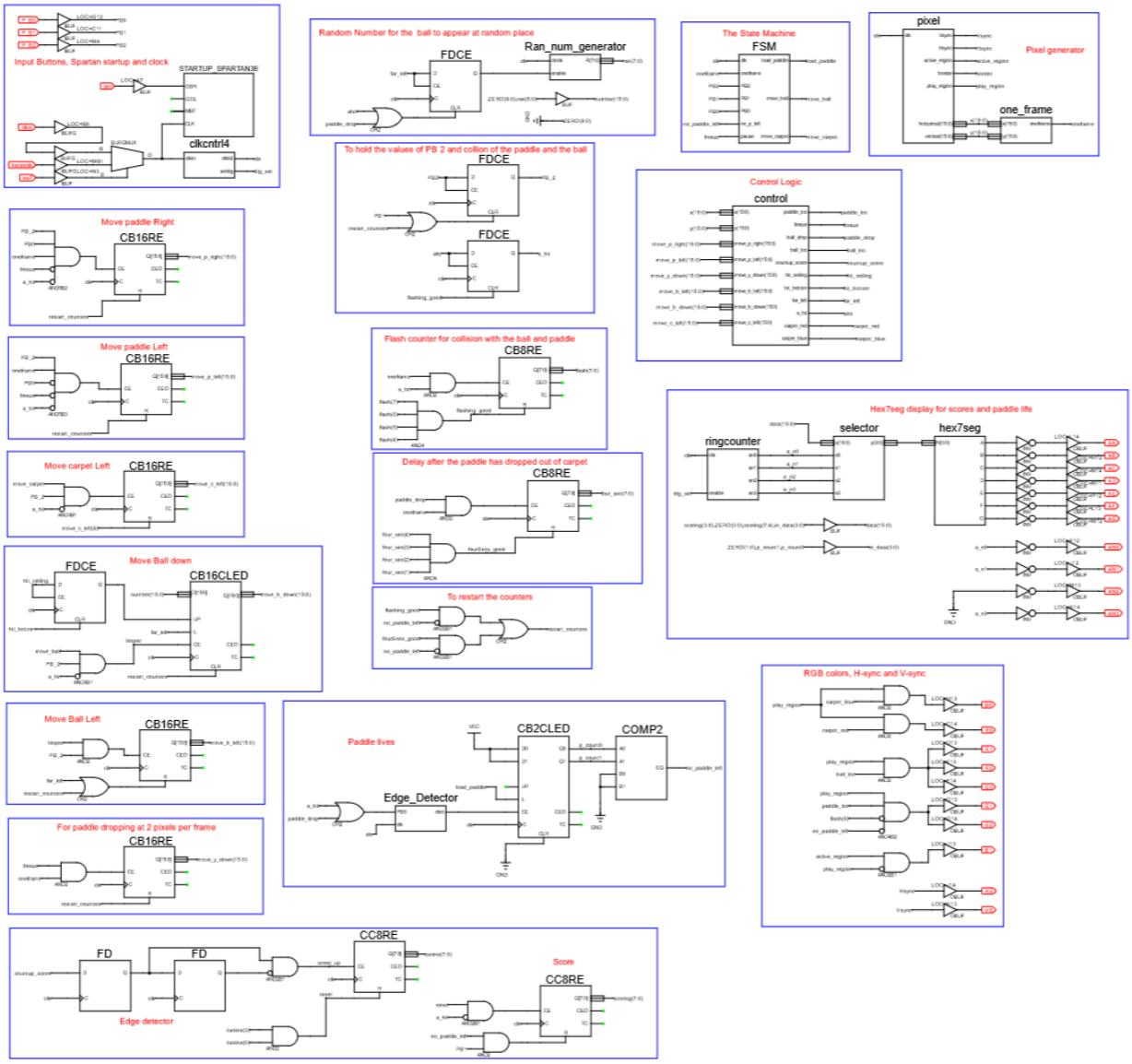

Figure 11: top level implementation

For clarity, I included individual components on the top level implementation.

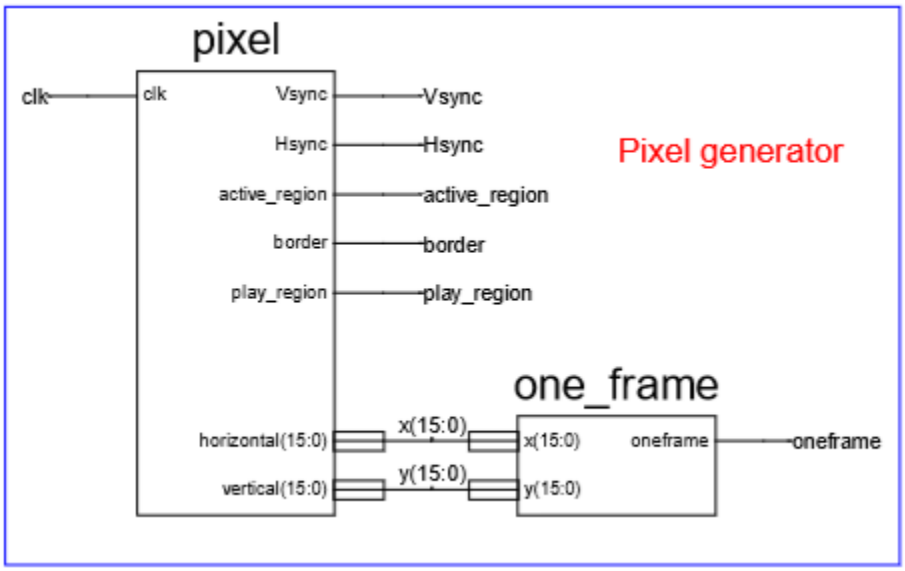

Figure 12: Implementing the active region, play region, as well as the signal that goes high only once every frame

VGAcontrol.v — Thu Mar 10 21:14:26 2016

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    16:49:28 02/23/2016
7   // Design Name:
8   // Module Name:    VGAcontrol
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 VGAcontrol(
22      input [15:0] horizontal,
23      input [15:0] vertical,
24      output Vsync,
25      output Hsync,
26      output reset_hori,
27      output reset_verti,
28      output active_region,
29      output border,
30      output play_region
31      );
32
33     assign Vsync = vertical<489 | vertical>490;
34     assign Hsync = horizontal<655 | horizontal>750;
35
36     assign active_region = horizontal<=639 & vertical<=479;
37     assign border = horizontal<8 | (horizontal>631 & horizontal<=640) | vertical<8 | (
    vertical>471 & vertical<480); // 632 & 639? 472 and 479?
38     assign play_region = horizontal>8 & horizontal<632 & vertical>255 & vertical<472;
39
40     assign reset_hori = horizontal == 799;
41     assign reset_verti = horizontal == 799 & vertical == 524;
42
43  endmodule
44
```

Figure 13: Defining the play region, active region, H sync and V sync

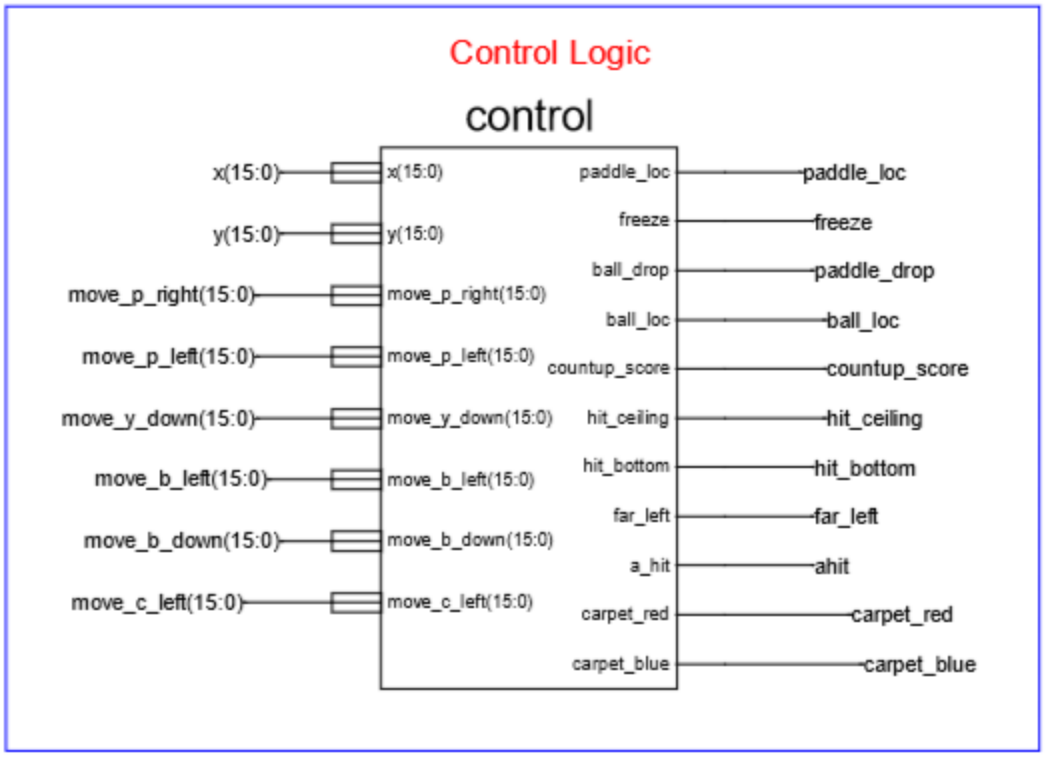

Figure 14: The control logic of the locations of ball, carpet, and paddle, and other signals

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    20:47:19 03/03/2016
7   // Design Name:
8   // Module Name:    control
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 control(
22     input [15:0] x,
23     input [15:0] y,
24     input [15:0] move_p_right, // move paddle right
25     input [15:0] move_p_left,  // or move paddle left
26     input [15:0] move_y_down,   // move paddle down when falls
27     //input [5:0] random_num,
28     input [15:0] move_b_left,    // move ball left vertically
29     input [15:0] move_b_down,    // move ball down horizontally
30     input [15:0] move_c_left,    // move carpet left
31
32     output paddle_loc,
33     output freeze,
34     output ball_drop,
35     output ball_loc,
36     output countup_score,
37     output hit_ceiling,
38     output hit_bottom,
39     output far_left,
40     output a_hit,
41     output carpet_red,
42     output carpet_blue
43      );
44
45     wire carpet_region_x = (x>= 55 & x <=583);
46     wire carpet_region_y = (y>=400 & y<= 415);
47     wire paddle_loc_y = y>= 337 + (2*move_y_down) & y<= 352 + (2*move_y_down) | y>=384
   + (2*move_y_down) & y<= 399+ (2*move_y_down);
48     wire ball_loc_y = (y>=(256 + move_b_down)) & (y<=(263 + move_b_down));
49
50     assign paddle_loc = (x>=239 + move_p_right - move_p_left & x<= 255+ move_p_right -
   move_p_left) & paddle_loc_y;
51     assign freeze = 239 + move_p_right - move_p_left <=38 | 255 +move_p_right -
   move_p_left >=600;    // maybe 55 and 583 at the edge
52     assign ball_drop = 337 +(2*move_y_down) >= 471; //maybe 472
53     assign ball_loc = (x>= 623 - move_b_left & x<=631 -move_b_left) & ball_loc_y;
```


Figure 15: Verilog of the Controlling logic part 1

```
54
55      wire countup_y = (y>=353 & y<=383) & (y>=262 + move_b_down) & y<=(262 + move_b_down
    );
56      wire countup_x = (x>=240+ move_p_right - move_p_left & x<= 250+ move_p_right -
    move_p_left) & (x>= 623 - move_b_left & x<=631 -move_b_left);
57
58      //wire countup_x = (x>=245+ move_p_right - move_p_left & x<= 250+ move_p_right -
    move_p_left) & (x>= 623 - move_b_left & x<=631 -move_b_left);
59      //assign countup_score = (x==624 - move_b_left) & ball_loc_y &
    (x==255+move_p_right - move_p_left ) & (y >=353 & y<= 383); //y <=353 & y<= 383
60
61      assign countup_score = countup_y & countup_x;
62
63      assign hit_ceiling = 256 + move_b_down <= 256; //  maybe <=
64      assign hit_bottom = (263 +move_b_down) >= 400 & carpet_region_y; // maybe >=  was
    carpet_region
65      assign far_left = 631 - move_b_left == 7; // maybe <=
66
67      assign a_hit = paddle_loc & ball_loc;
68
69      wire [15:0] carpet_1 = x + move_c_left;
70
71      assign carpet_red = carpet_1[5] & carpet_region_x & carpet_region_y;  // mtnf
72      assign carpet_blue = ~carpet_1[5] & carpet_region_x & carpet_region_y;  //mntf
73
74  endmodule
75
```

Figure 16: Controlling logic part 2

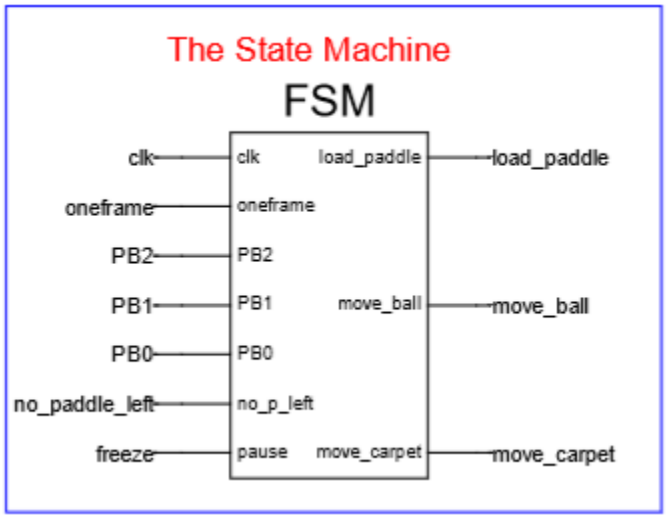

Figure 17: The finite state machine

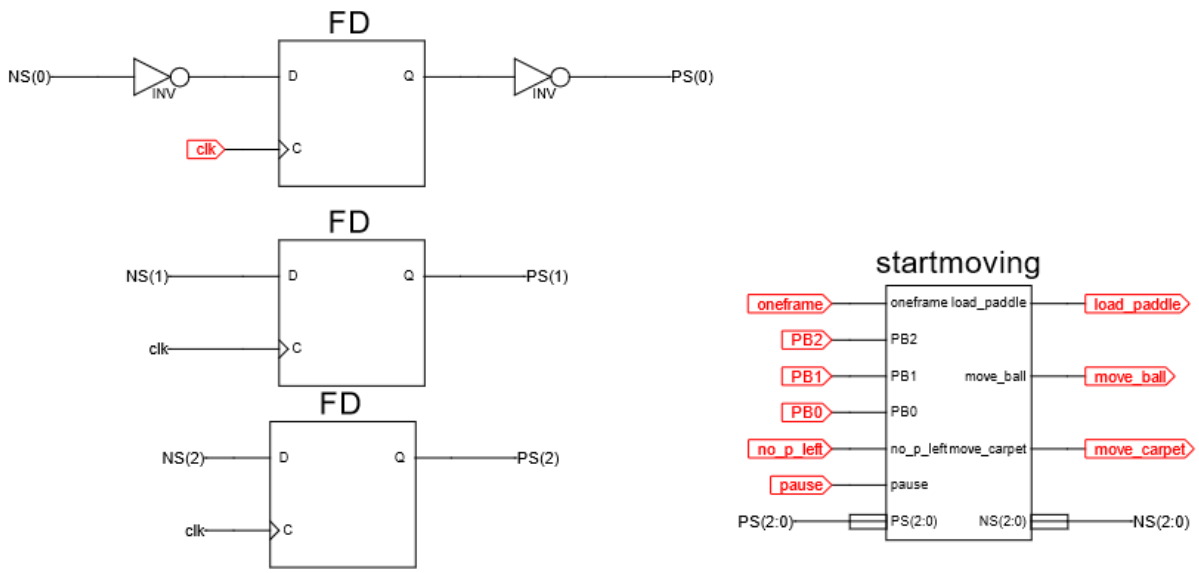

Figure 18: Finite state implementation

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    23:27:27 03/03/2016
7   // Design Name:
8   // Module Name:    start_moving
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 startmoving(
22     input oneframe,
23     input PB2,
24     input PB1,
25     input PB0,
26     input no_p_left,
27     input pause,
28  // input flashing_good,
29     input [2:0] PS,    // not [0:2] !!!
30
31     output [2:0] NS,
32     output load_paddle,
33     output move_ball,
34     output move_carpet
35      );
36     wire idle, start, lost;
37     wire n_idle, n_start, n_lost;
38
39     assign idle = PS[0];
40     assign start = PS[1];
41     assign lost = PS[2];
42
43     assign NS[0] = n_idle;
44     assign NS[1] = n_start;
45     assign NS[2] = n_lost;
46
47     assign n_idle = ~PB2 & idle | lost& PB1 ;
48     assign n_start = PB2 & idle | start & ~no_p_left;
49     assign n_lost = start&no_p_left | lost& ~PB1 ;
50
51     assign move_carpet = oneframe & start &~pause| lost&oneframe;   // ooneframe &
   start | oneframe& lost
52     assign load_paddle = idle&~PB2;
53     assign move_ball = lost&oneframe | oneframe&start & ~pause ;   // might need to
   add    //oneframe&PB2
54
55  endmodule
```


Figure 19: State machine

one_frame.v Thu Mar 10 21:13:00 2016

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    18:47:43 02/29/2016
7   // Design Name:
8   // Module Name:    one_frame
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 one_frame(
22      input [15:0] x,
23      input [15:0] y,
24      output oneframe
25      );
26
27     assign oneframe = x== 799 & y==524;
28  endmodule
29
```

Figure 20: We need a signal that goes high once every frame, so this is the implementation of it

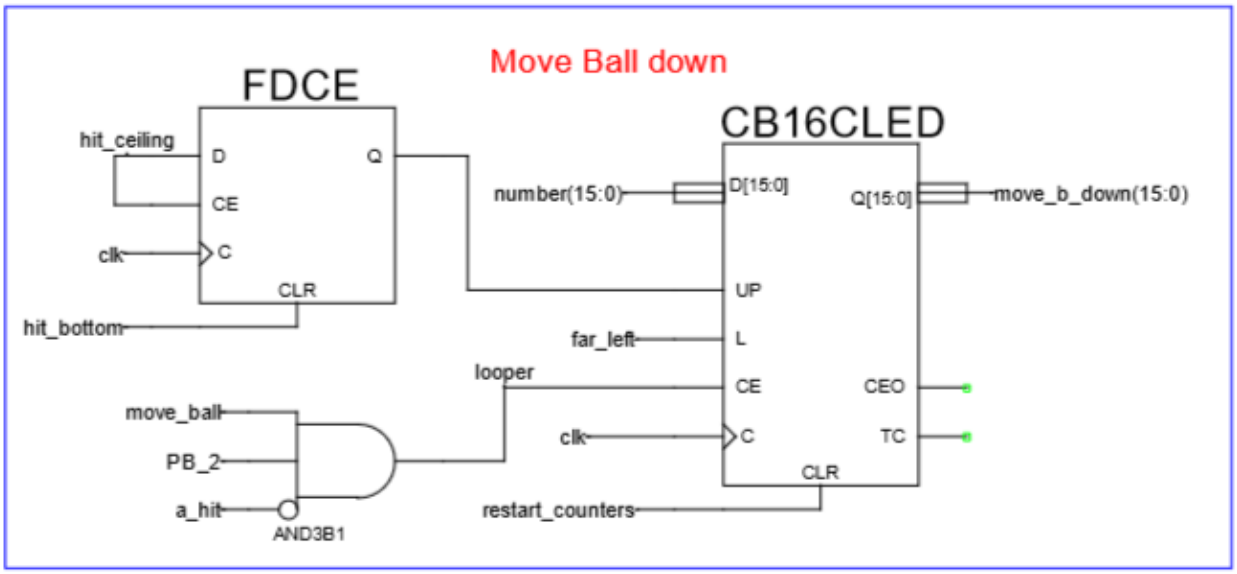

Figure 21: To move the ball up or down, using the up-down counter. We don't have to worry about 2's complements as we are directly loading the value

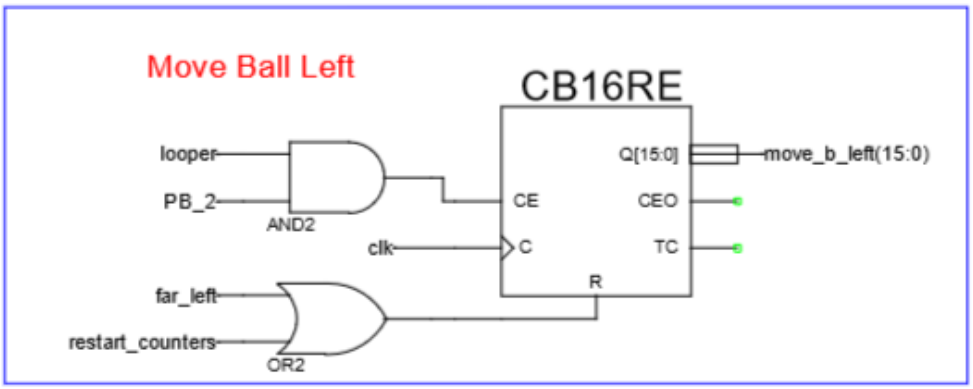

Figure 22: To move the ball to the left in x direction

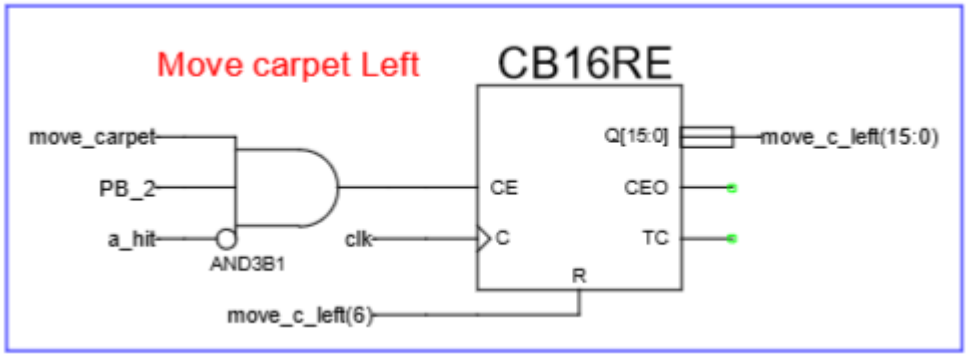

Figure 23: To move the carpet to the left, we used 5th bit to get the 32 pixel wide and reset every time we have the 6th bit goes high.

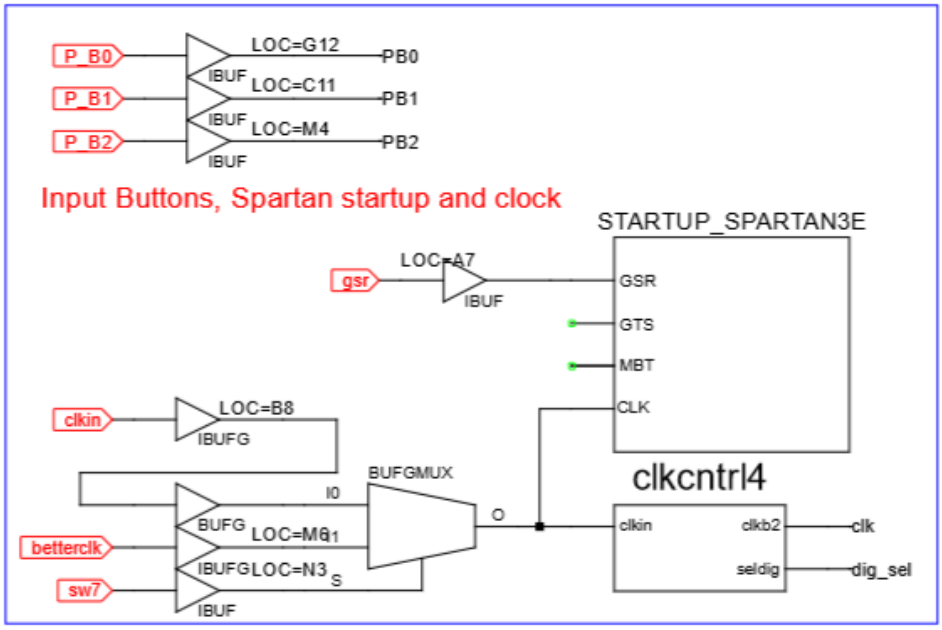

Figure 24: The Startup of the FPGA board and the clock

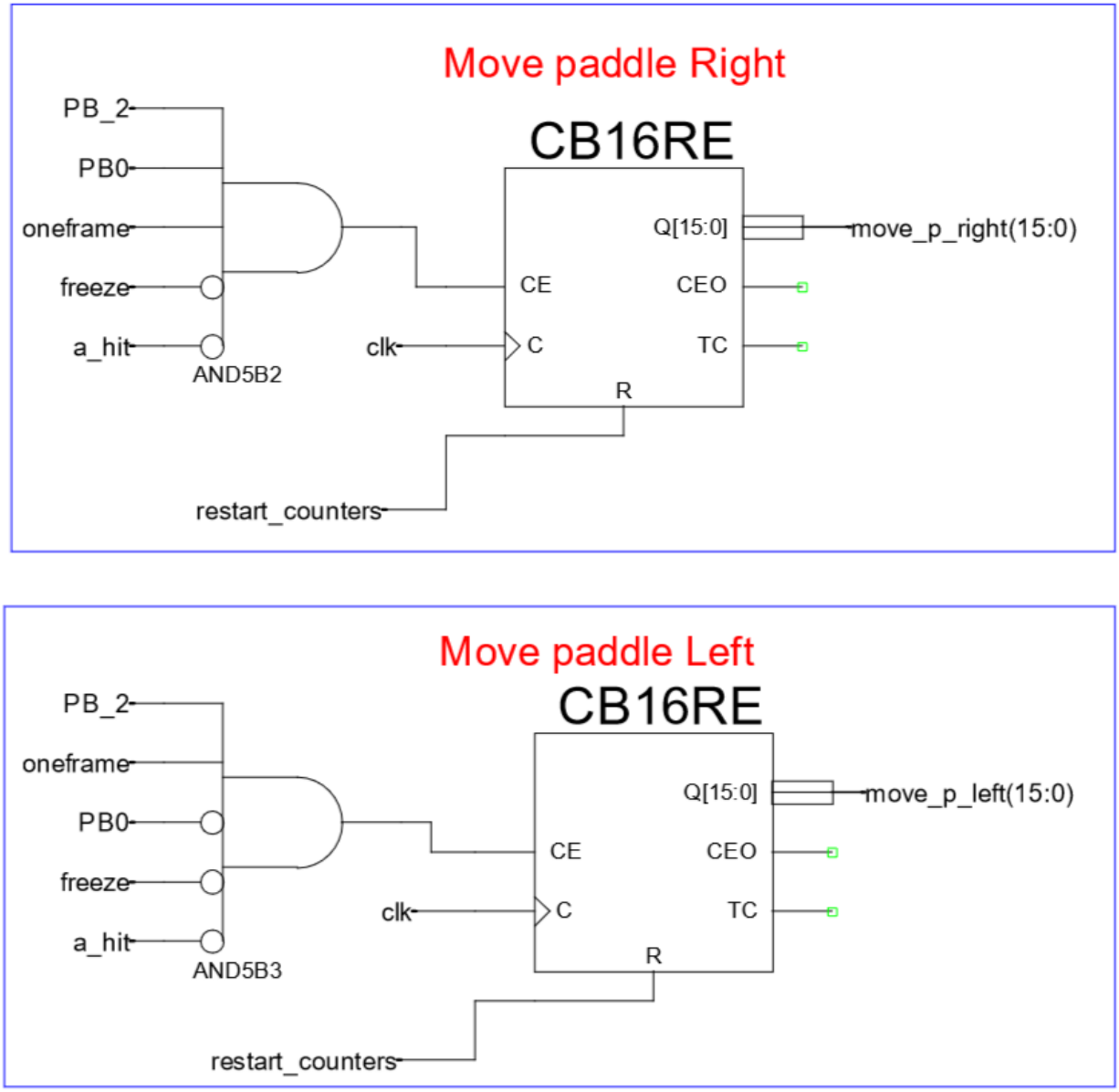

Figure 25: To move the paddle left and right

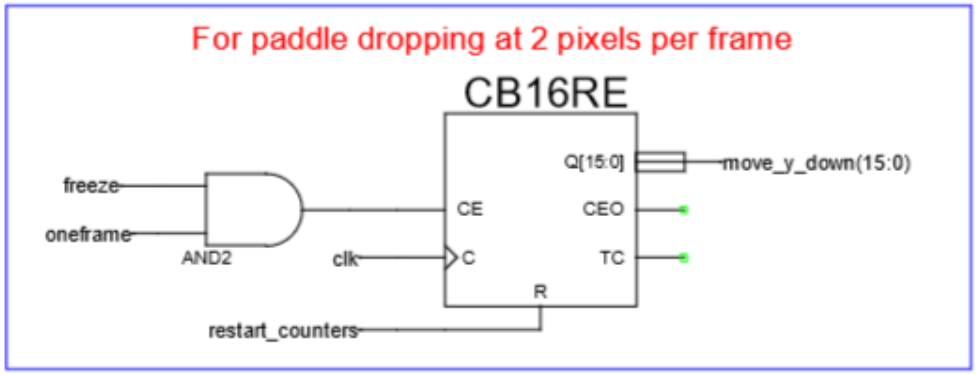

Figure 26: To drop the paddle at 2 pixels per frame

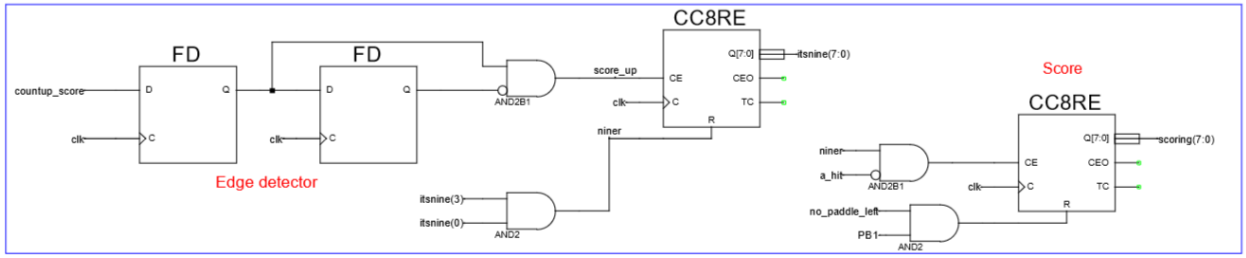

Figure 27: The score implementation

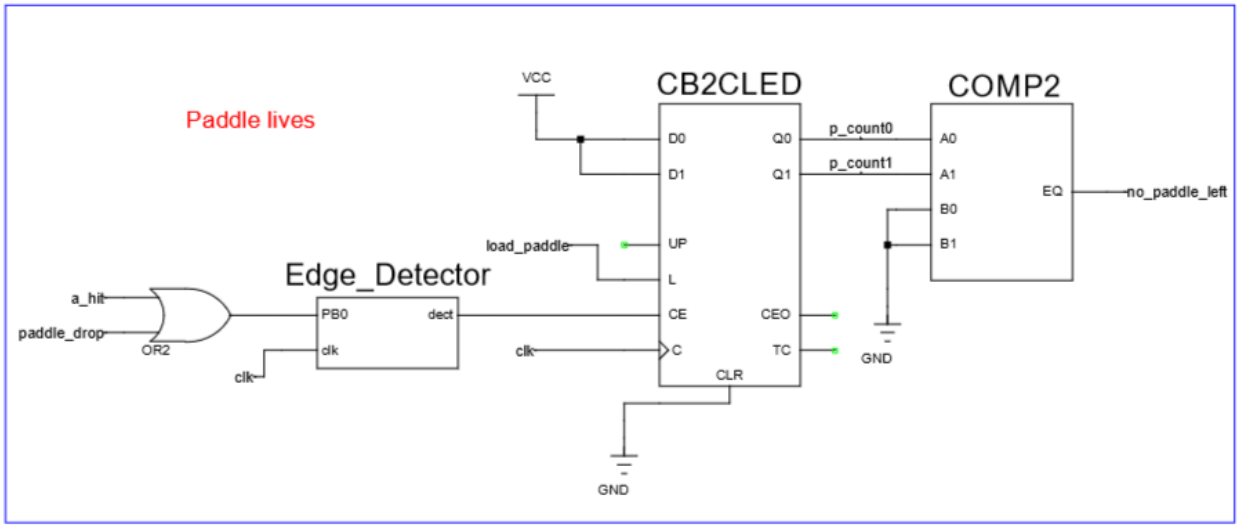

Figure 28: This shows the paddle lives and output goes high if we don't have any more paddle left

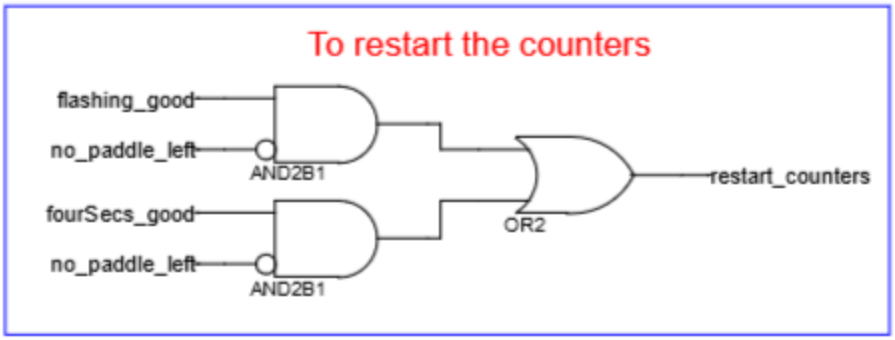

Figure 29: To restart the game or get back to the initial position

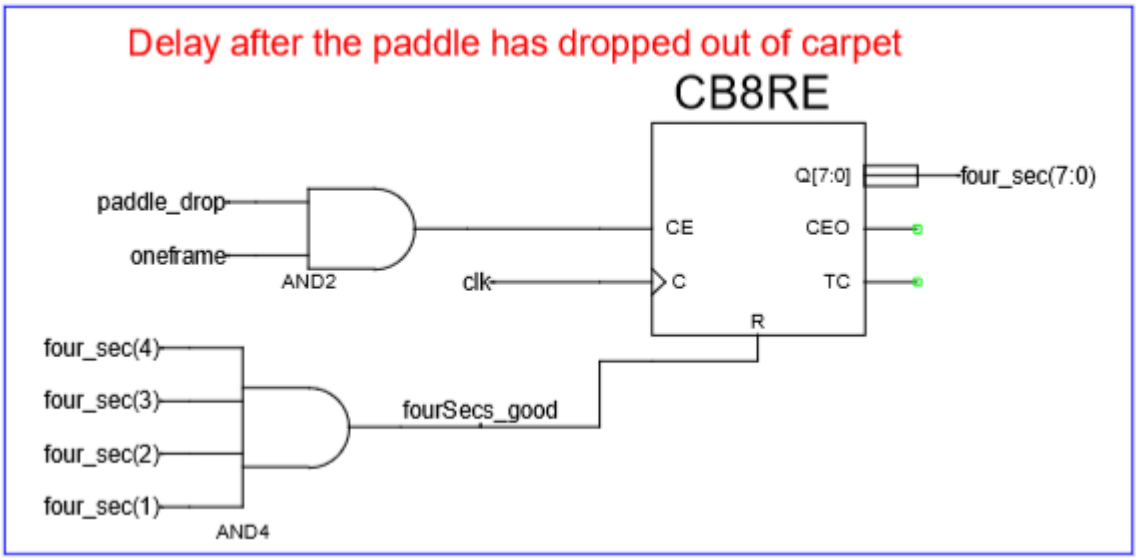

Figure 30: I Added a bit of delay after the paddle has dropped

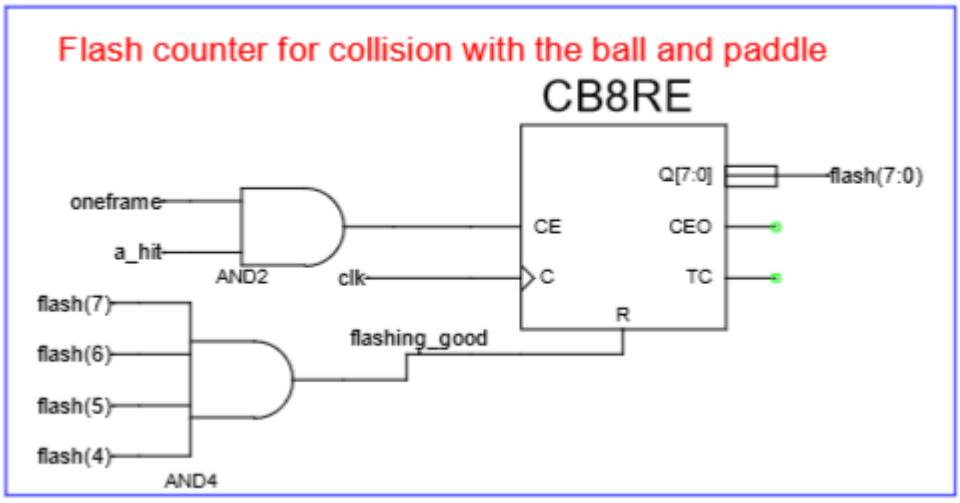

Figure 31: Flashing of the paddle after the collision with the ball

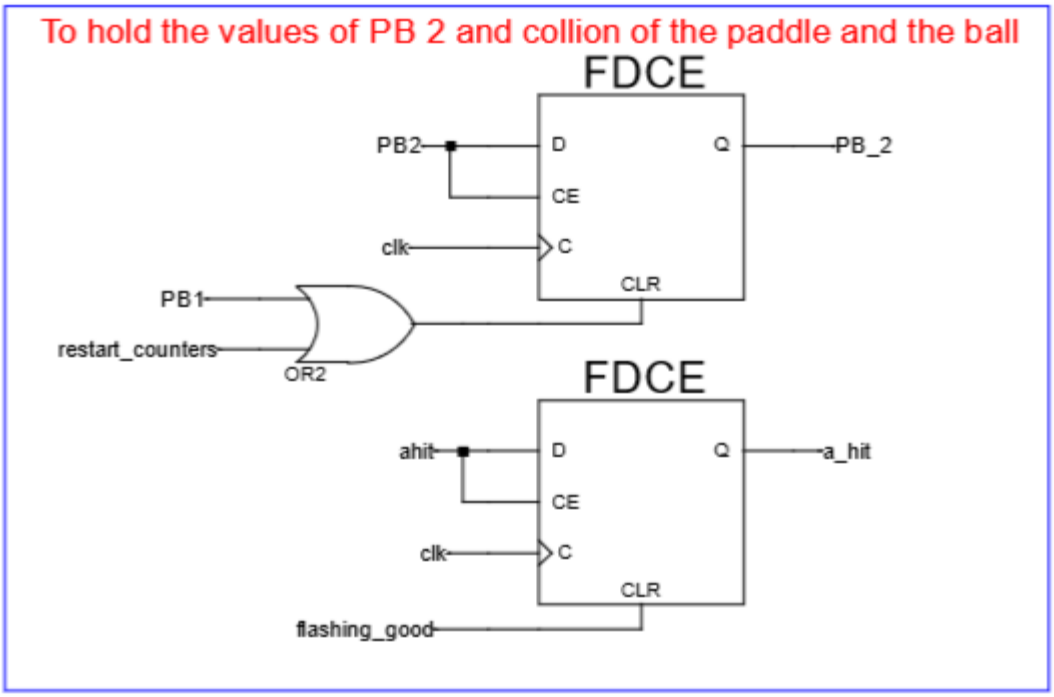

Figure 32: To hold values, I used flip flop with the CE and D, and CLR.

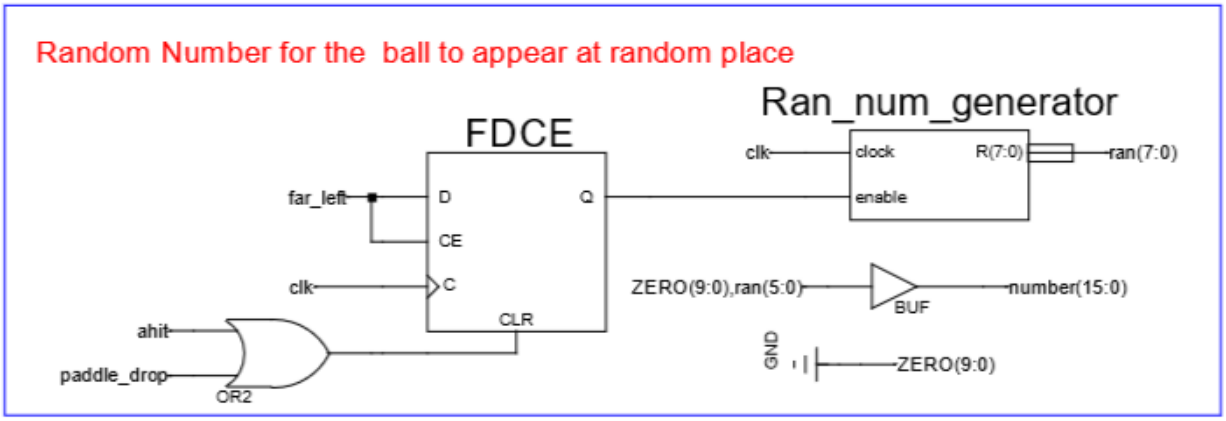

Figure 33: to start the ball at random position

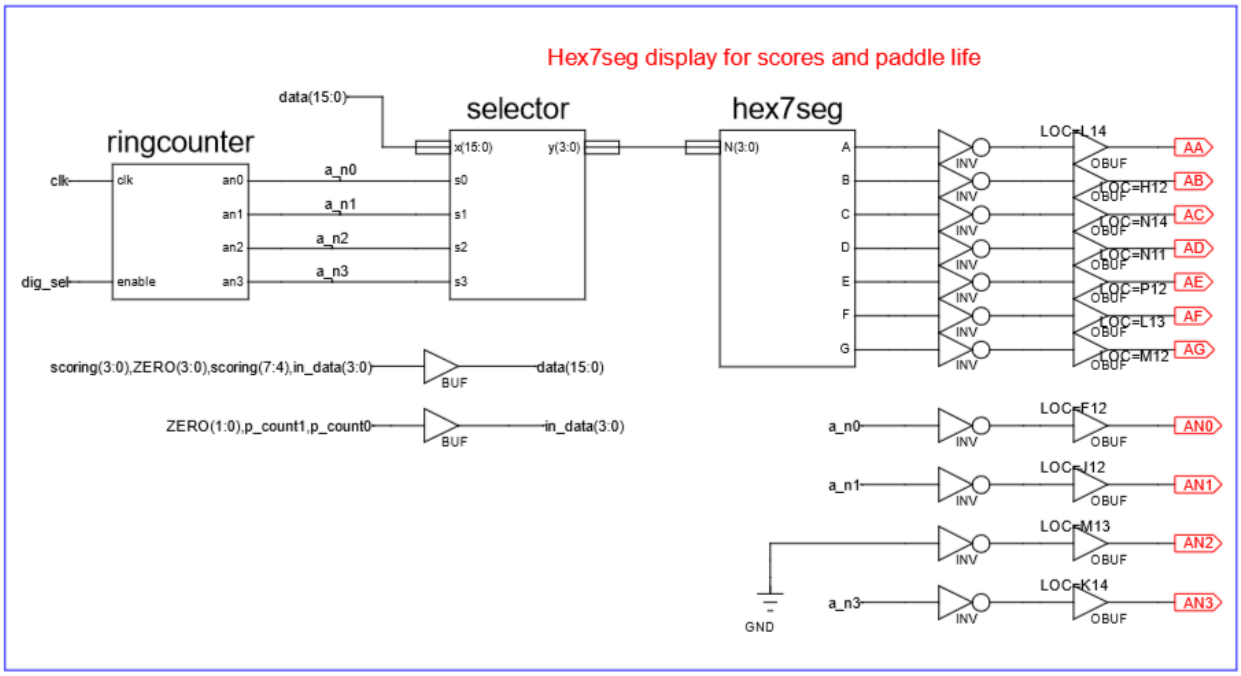

Figure 34: Hex7seg top level

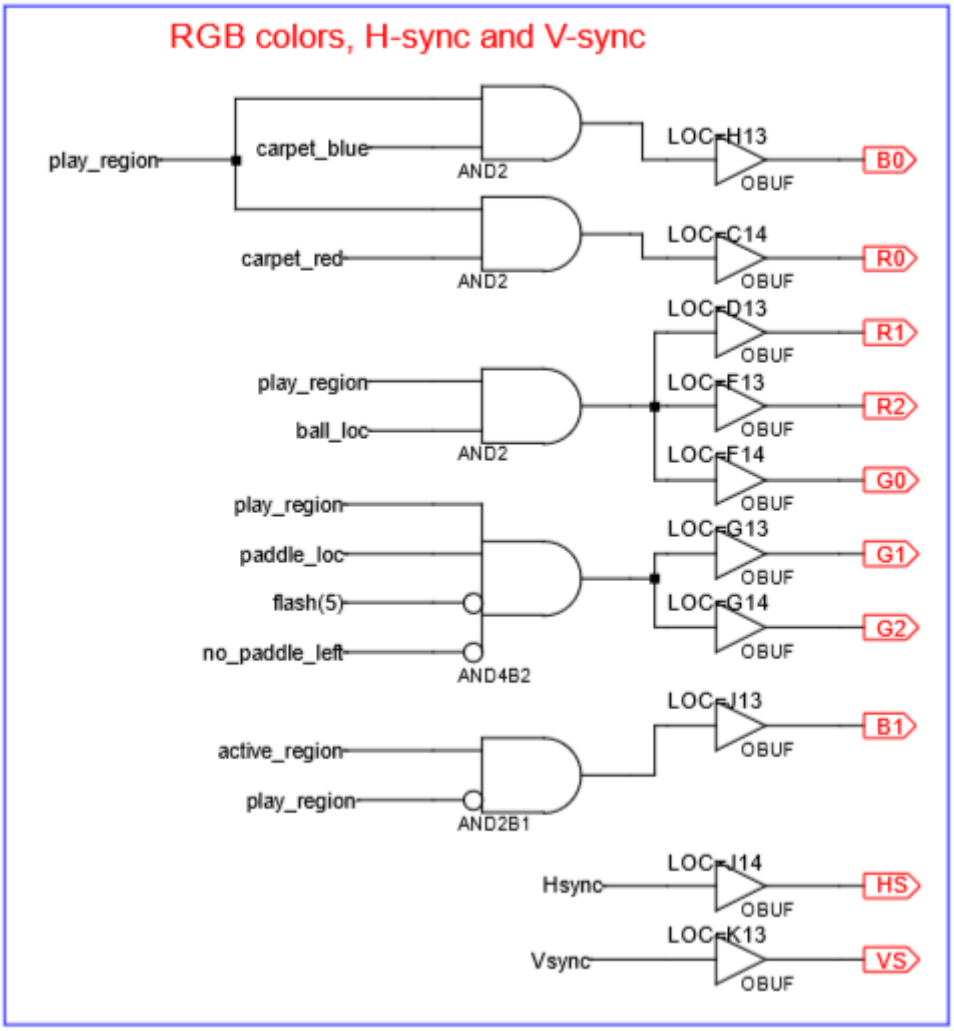

Figure 35: The color bits and H-sync and V-sync

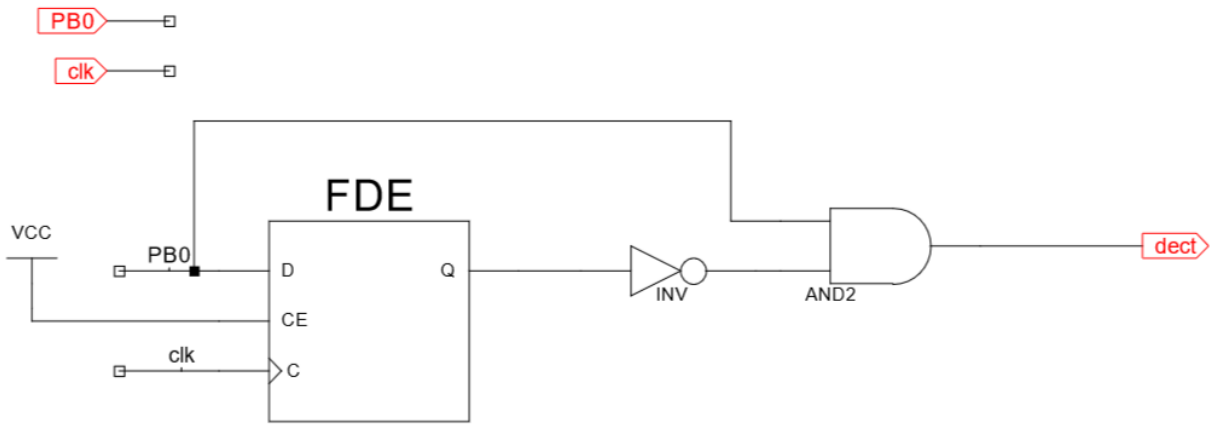

Figure 36: Edge detector from previous lab

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date:    12:36:32 01/28/2016
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 A,
24      output B,
25      output C,
26      output D,
27      output E,
28      output F,
29      output G
30      );
31
32     wire [5:0] sel;
33     or ( A, sel[5], sel [4], sel [3], sel [2], sel [1], sel [0]);
34     and (sel[5], ~N[2], ~N[0]);
35     and (sel[4], ~N[3], N[1]);
36     and (sel[3], N[2], N[1]);
37     and (sel[2], N[3], ~N[0]);
38     and (sel[1], ~N[3], N[2], N[0]);
39     and (sel[0], N[3], ~N[2], ~N[1]);
40
41     wire [4:0] selB;
42     or (B, selB[4], selB[3],selB[2],selB[1],selB[0]);
43     and (selB[4], ~N[3], ~N[2]);
44     and (selB[3], ~N[2], ~N[0]);
45     and (selB[2], ~N[3], ~N[1], ~N[0]);
46     and (selB[1], ~N[3], N[1], N[0]);
47     and (selB[0], N[3], ~N[1], N[0]);
48
49     wire [4:0] selC;
50     or (C, selC[4], selC[3],selC[2], selC[1], selC[0]);
51     and (selC[4], ~N[3], ~N[1]);
52     and (selC[3], ~N[3], N[0]);
53     and (selC[2], ~N[1], N[0]);
54     and (selC[1], ~N[3], N[2]);
55     and (selC[0], N[3], ~N[2]);
56
57     wire [4:0] selD;
```


Figure 37: Hex7seg part 1

```
58      or (D, selD[4], selD[3],selD[2], selD[1], selD[0]);
59      and (selD[4], N[3], ~N[1]);
60      and (selD[3], ~N[3], ~N[2], ~N[0]);
61      and (selD[2], ~N[2], N[1], N[0]);
62      and (selD[1], N[2], ~N[1], N[0]);
63      and (selD[0], N[2], N[1], ~N[0]);
64
65      wire [3:0] selE;
66      or (E, selE[3],selE[2], selE[1], selE[0]);
67      and (selE[3], ~N[2], ~N[0]);
68      and (selE[2], N[1], ~N[0]);
69      and (selE[1], N[3], N[1]);
70      and (selE[0], N[3], N[2]);
71
72      wire [4:0] selF;
73      or (F, selF[4], selF[3],selF[2], selF[1], selF[0]);
74      and (selF[4], ~N[1], ~N[0]);
75      and (selF[3], N[2], ~N[0]);
76      and (selF[2], N[3], ~N[2]);
77      and (selF[1], N[3], N[1]);
78      and (selF[0], ~N[3], N[2], ~N[1]);
79
80      wire [4:0] selG;
81      or (G, selG[4], selG[3],selG[2], selG[1], selG[0]);
82      and (selG[4], ~N[2], N[1]);
83      and (selG[3], N[1], ~N[0]);
84      and (selG[2], N[3], ~N[2]);
85      and (selG[1], N[3], N[0]);
86      and (selG[0], ~N[3], N[2], ~N[1]);
87
88  endmodule
89
```

Figure 38: Hex7seg part2

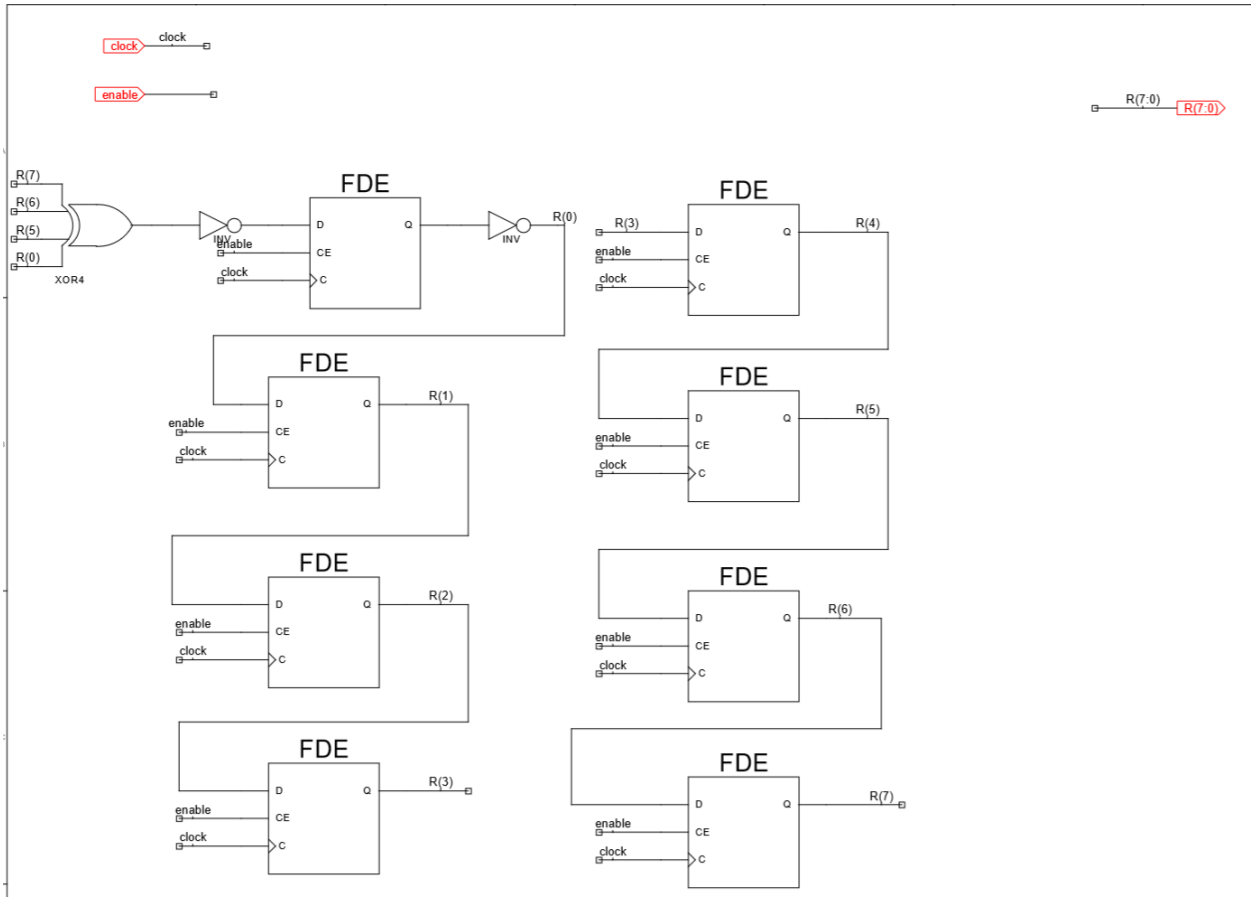

Figure 39: Random number generator from pervious lab

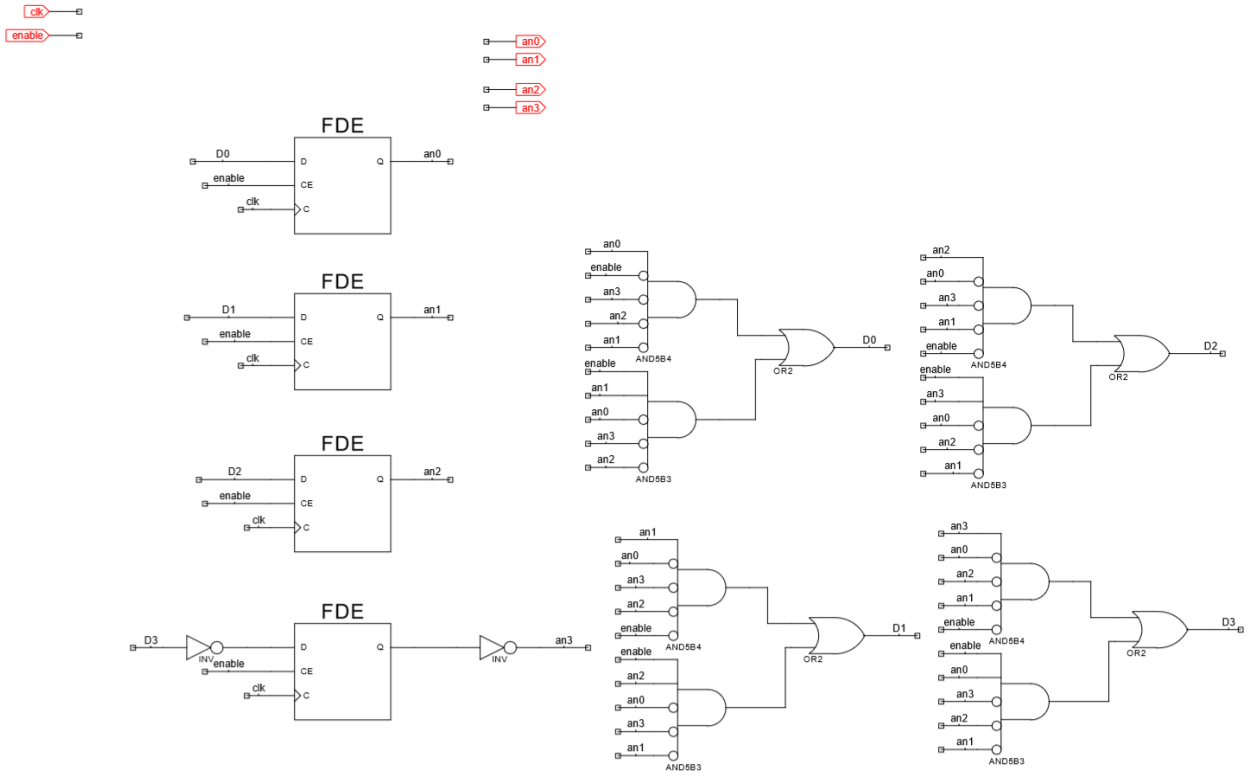

Figure 40: Ring counter

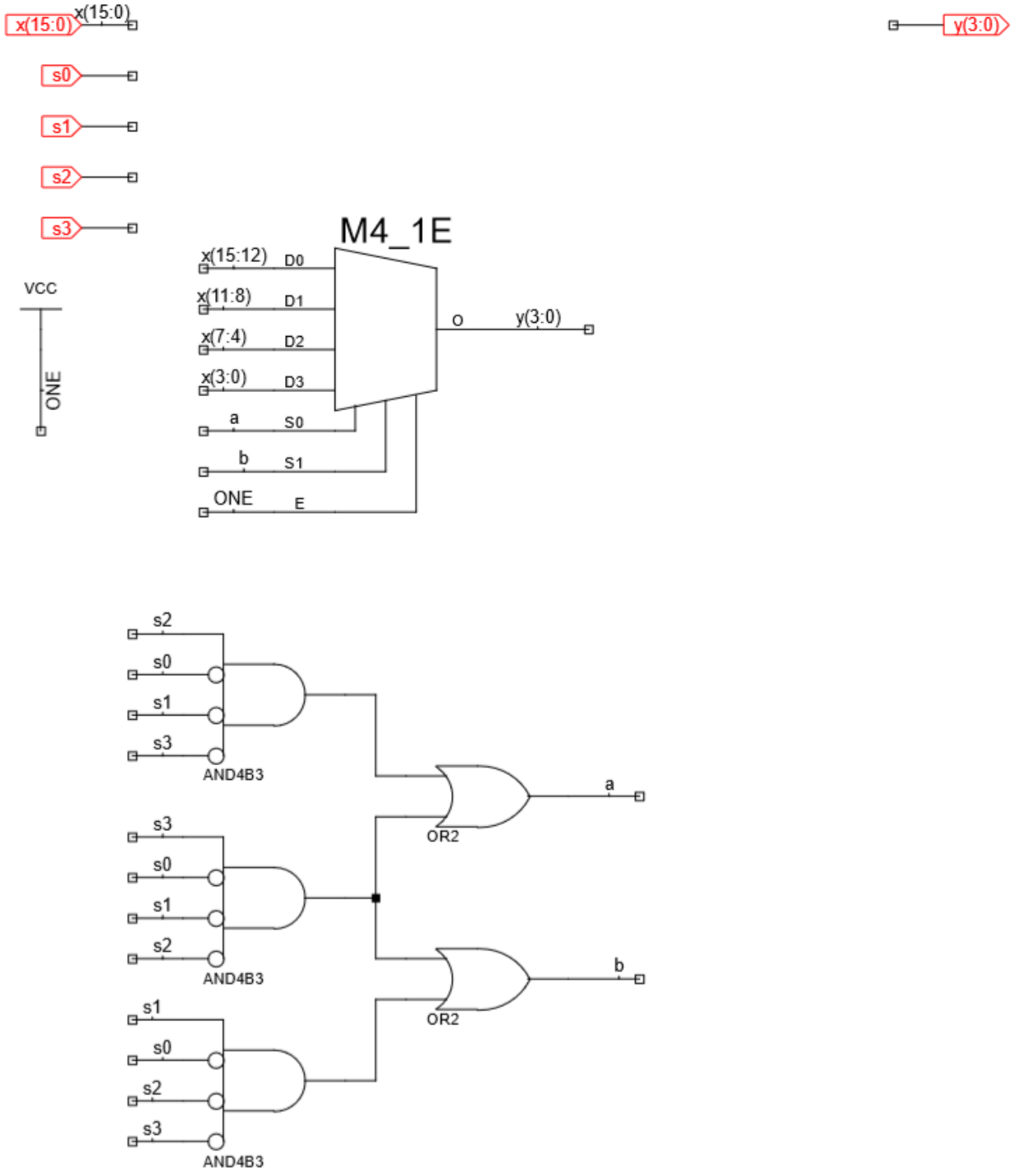

Figure 41: Selector from previous lab

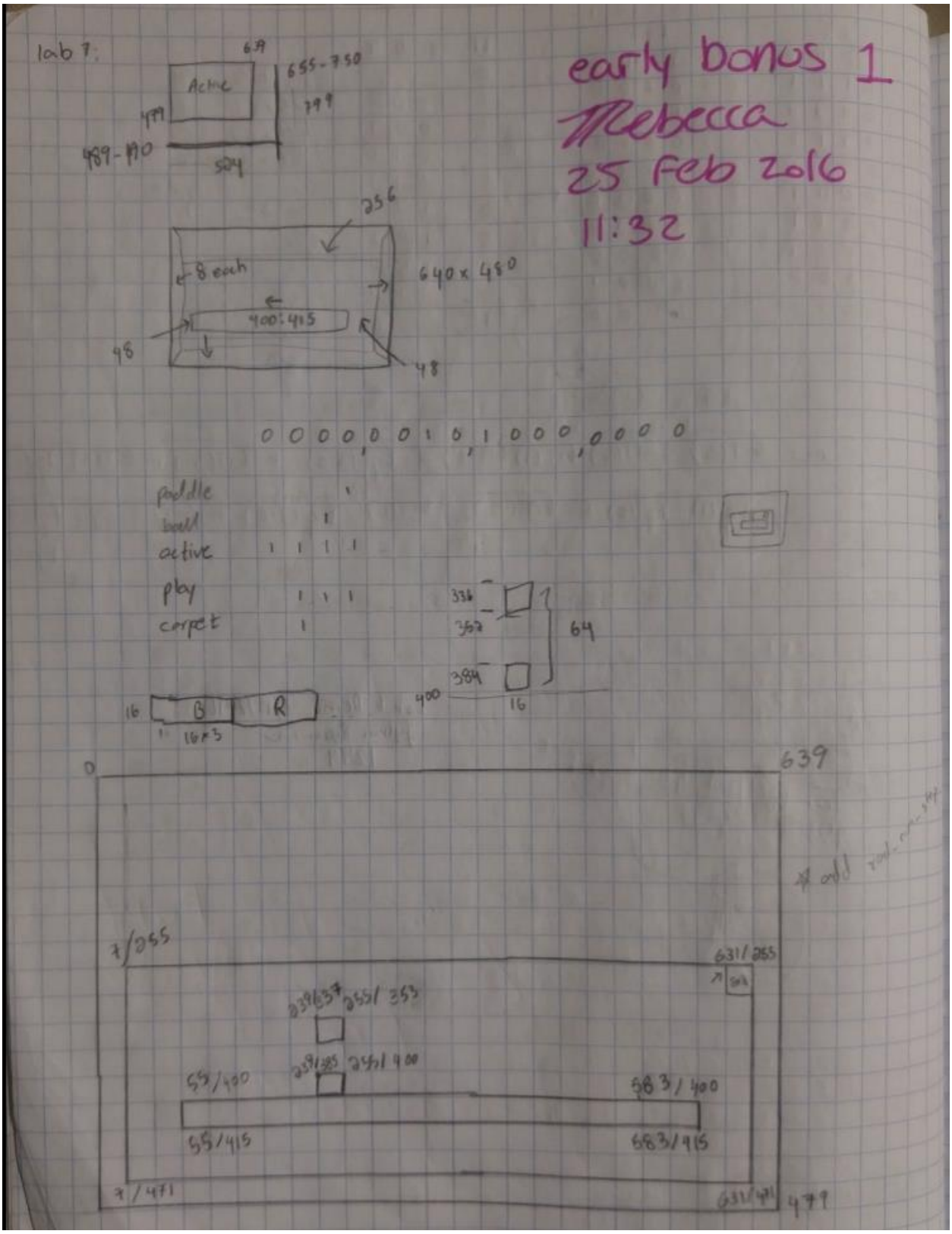

Figure 42: Lab notebook page 1

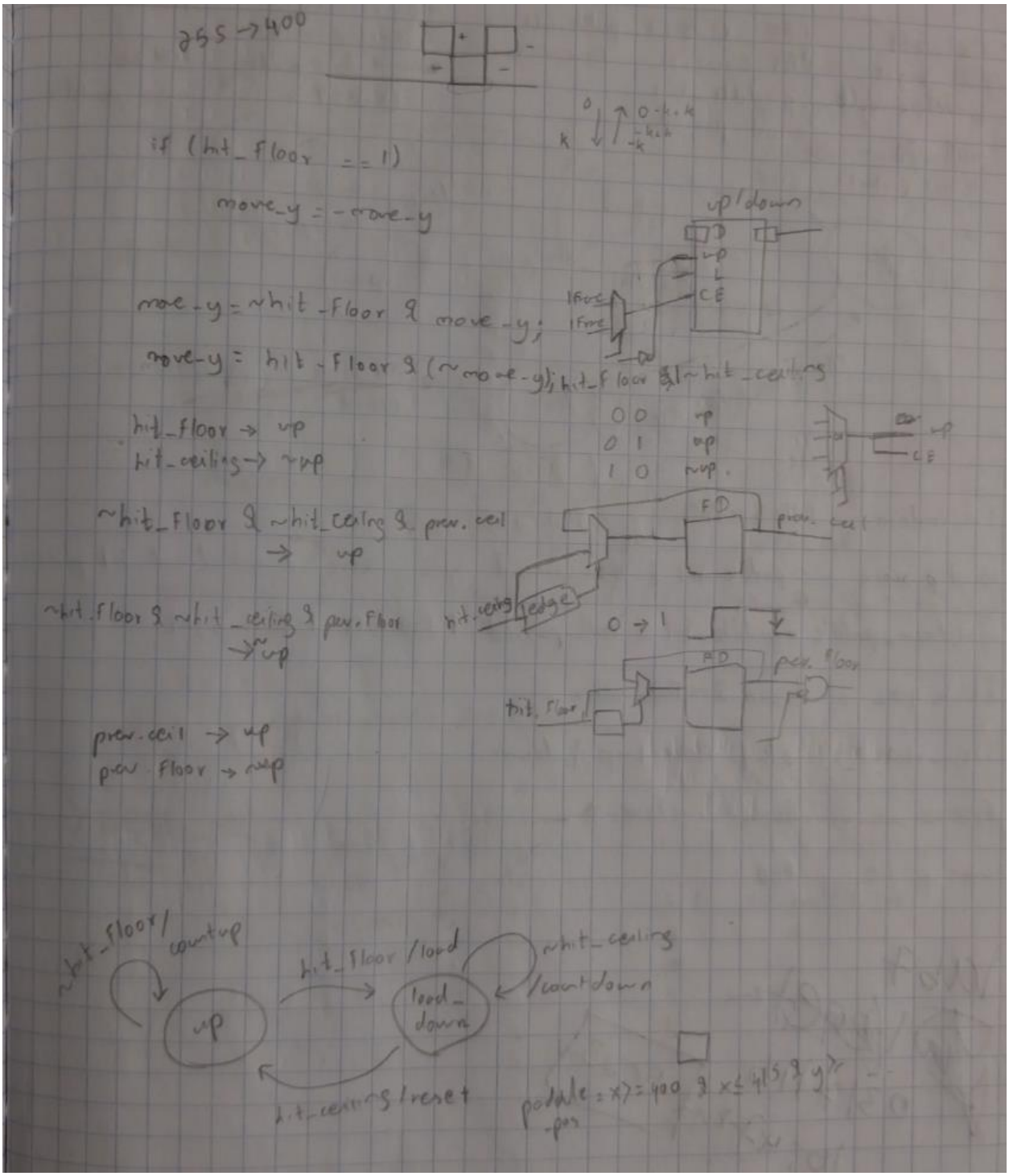

Figure 43: Lab notebook page 3

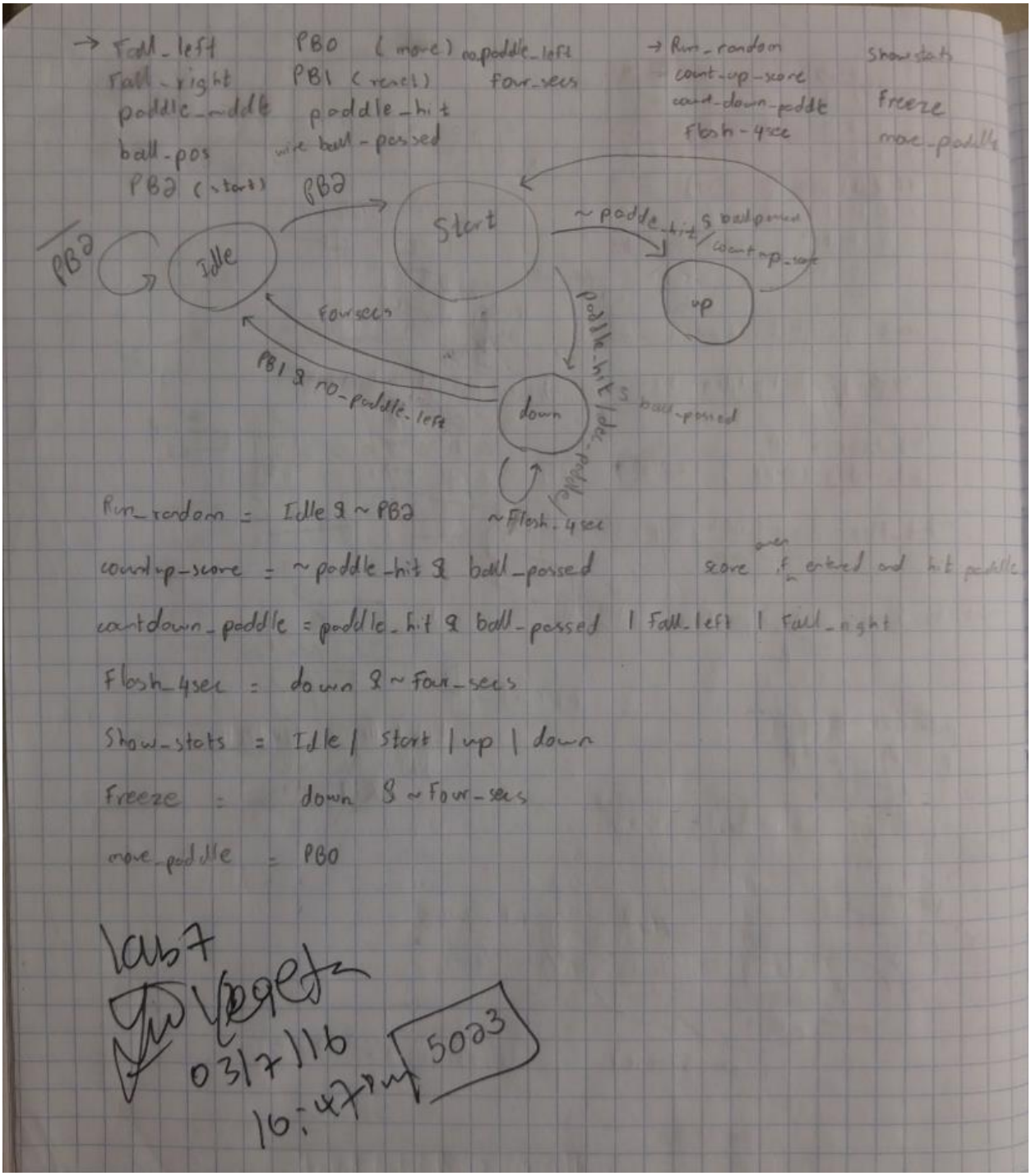

Figure 44: Lab notebook page 3

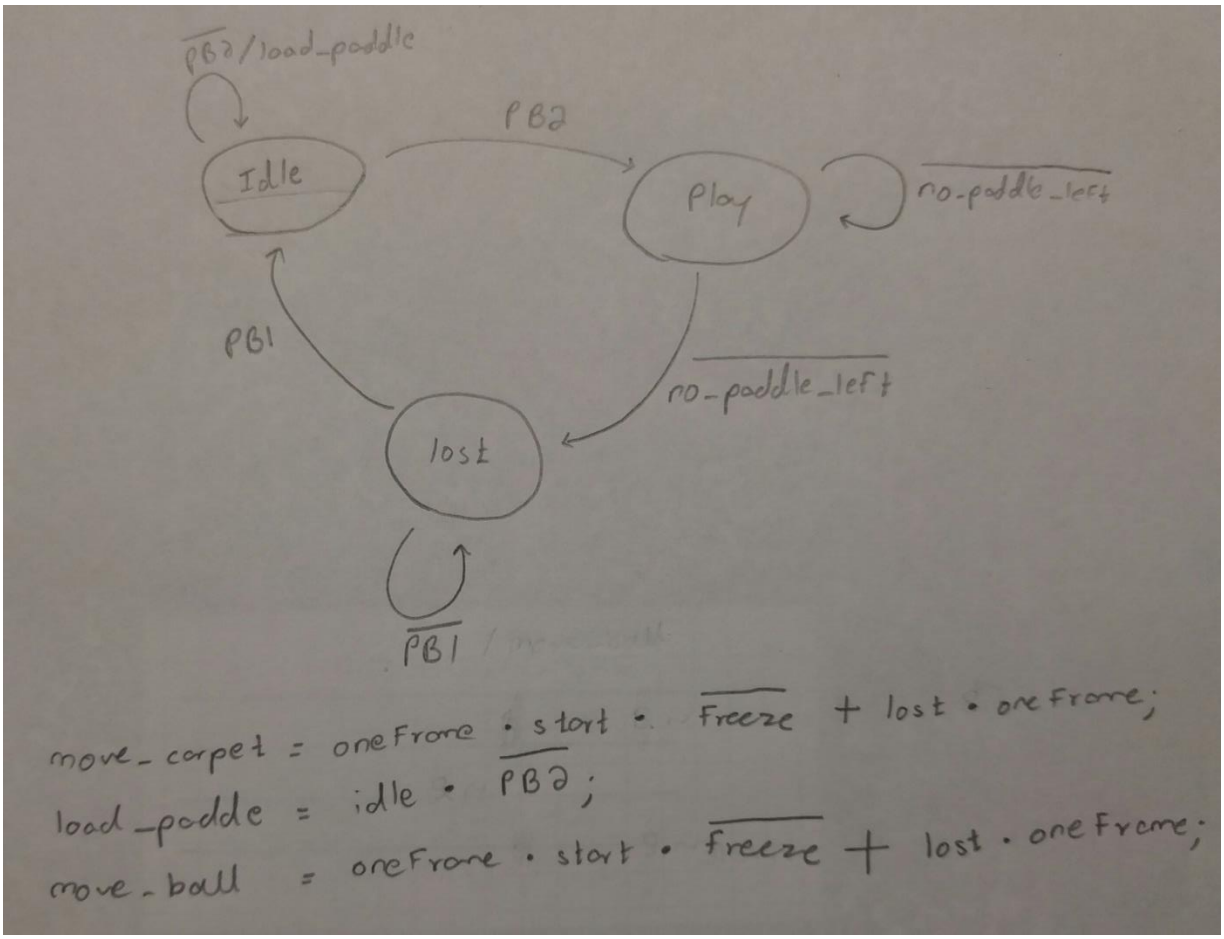

Figure 45: The state machine implementation