University of California, Santa Cruz

A Fly Swatting Game

Lab 7 Report

CMPE 100L - Spring 2015

Sargis S Yonan

syonan@ucsc.edu - 1370311

LAB SECTION: 3

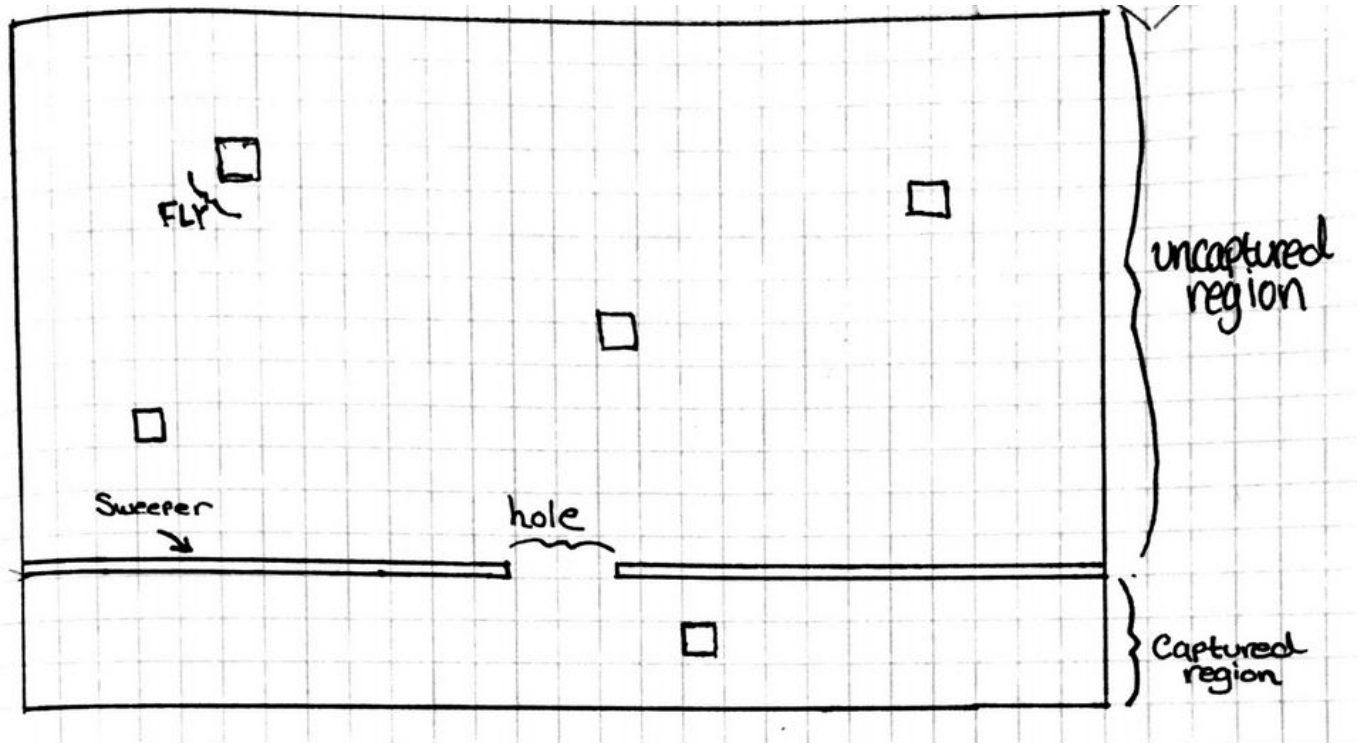
June 5th, 2015

Purpose

CMPE 100L, the lab for the Logic Design & Verilog course at UC Santa Cruz is infamous among engineering students within The Baskin School of Engineering mostly for one reason: Lab 7. Among being the most time consuming lab of the quarter, the lab was the keystone of the course tying together every major component of the class plus some. In this lab, I had to construct an arcade-style video game out of sequential circuits and logic. This was not something I was ever aware of being possible, and am shocked that it worked. The game, being a Fly Catching simulator, starts you out staring into the eyes of five frozen flies within a black void only with a mysteriously closed bar. After pressing the start button, a hole opens within the bar, and the flies begin to flash for four seconds. They proceed to fly all around the screen bouncing off of the walls. The objective of the game is to capture all five flies under the bar through the hole by moving the direction of the bar and avoiding letting the flies leave the pit-latrine captured region. This program, being a videogame, required video output from the FPGA board. This meant that I needed to configure the FPGA for VGA output to a monitor. From there I moved on to the logistics and control of the game. The Chinese philosopher Laozi

said, “Even the longest and most difficult ventures have a starting point”, and so the journey began

.



Prototyped drawing of the game play

Methods

Before I was able to start the game logic, I had to synchronize the horizontal and vertical components of the VGA output in order to display anything on the monitor. This was accomplished by beginning two counters for the H and V sync on initialization of the board to keep track of what pixel the display was currently writing to.

Synchronization

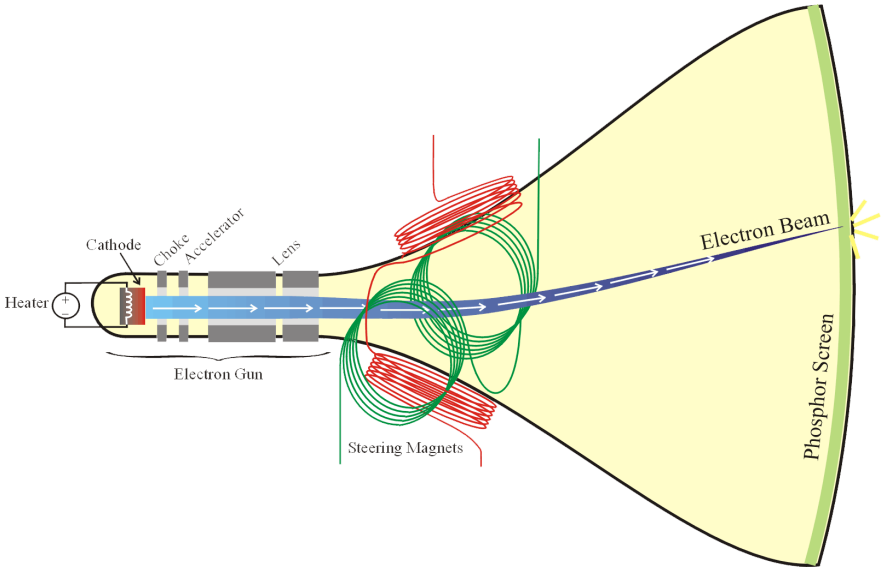


Photo Courtesy of The WikiMedia Foundation

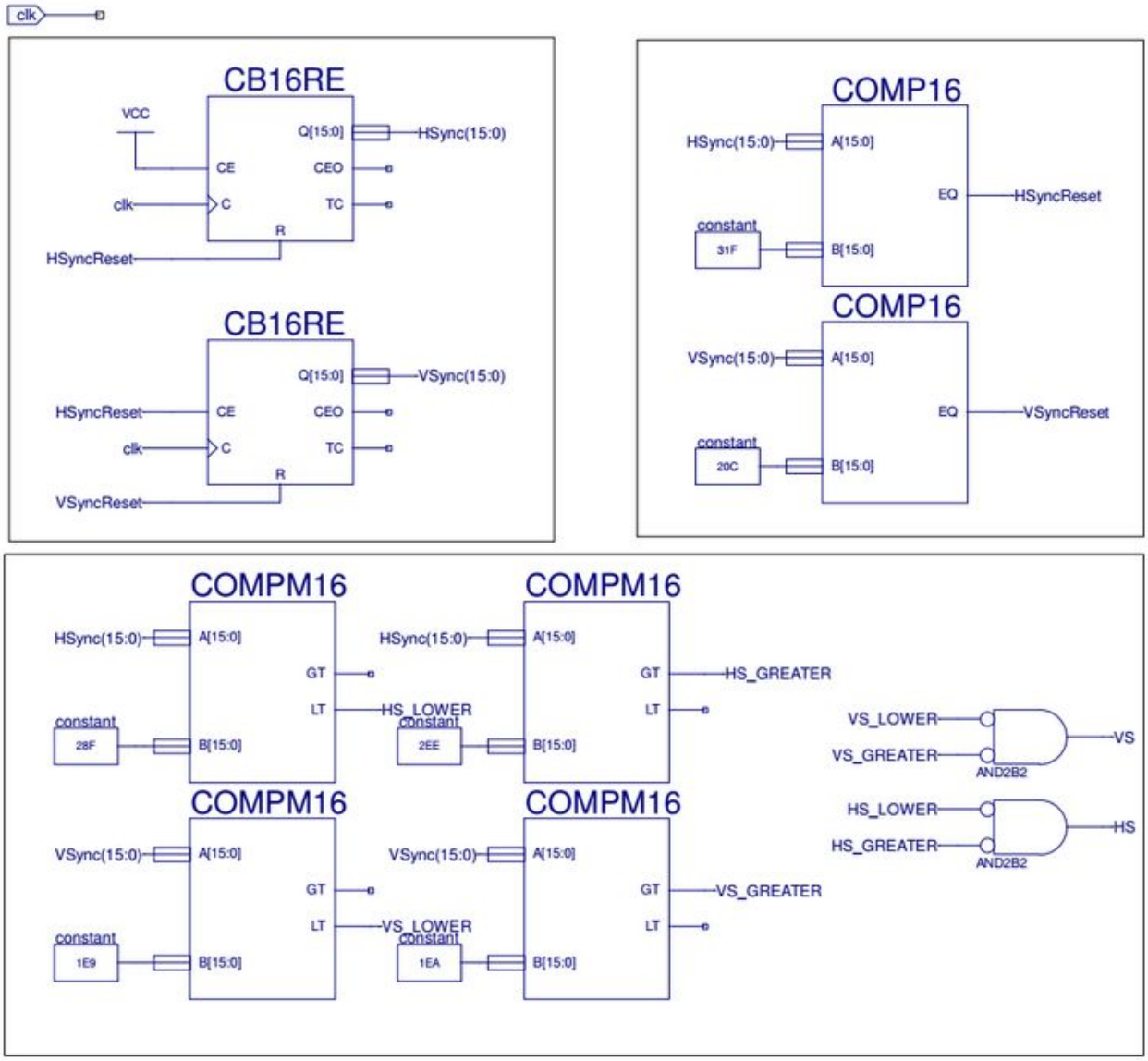
Old-fashioned cathode ray tube monitors ran electron beams across the screen in a matrix fashion by writing along the horizontal components from zero to the maximum horizontal length, then moving one pixel down vertically and repeating the horizontal write. This continued until the screen had been written through the maximum vertical length, where both counters were initialized to zero. This is called one frame refresh. Though the LCD display I used in lab was not CRT based, it still worked in the same way. This is due to the preserved conventions from the past of display technologies. Since the crystal oscillator on my board ran at 25MHz and VGA writes to a region of size 800 X 525 pixels, it took:

$$\frac{1}{25\ MHz} \times 800\ \times 525\ =\ .0168\ seconds$$

for one frame refresh, which is approximately a frame refresh rate of 60Hz (59.52 Hz), the industry standard timing for VGA displays in the United States.

Since my screen was only 640 X 480 pixels, I had to configure an active writing region to be bounded to the 640 X 480 size within the larger 800 X 525 size in order to know where to output color signals. This required outputting an ACTIVE signal to be high when HS and VS were counting within in this region.

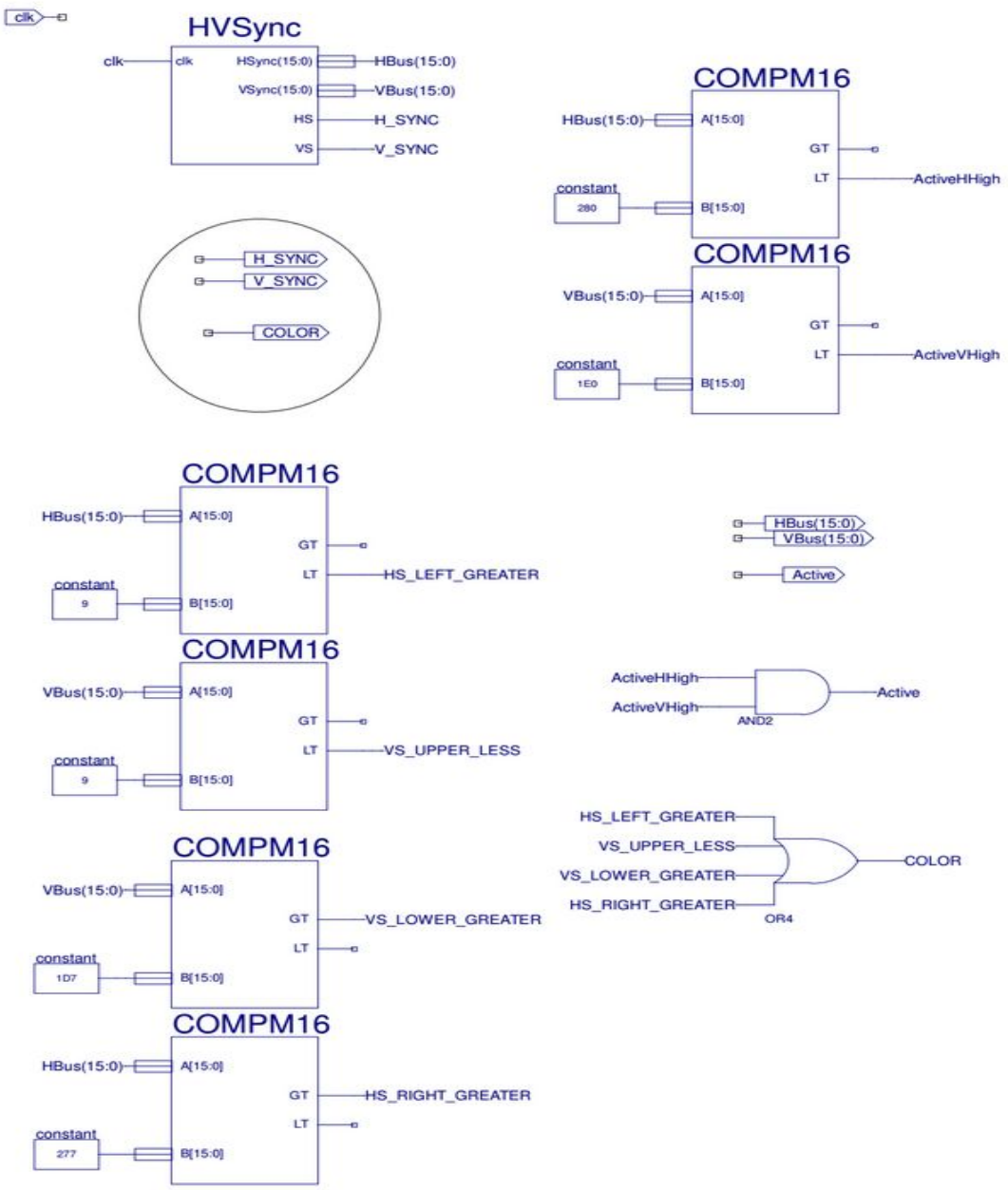
This was accomplished via the following method:



Horizontal and Vertical Syncing

Game Setting

After properly writing to the active region of the screen, I initiated more counters to keep track of more objects to write. I composed a border drawing module containing four counters that knew when the HS and VS signals and the active region were within the bounds in which I wanted to draw the 8 pixel wide borders. I then outputted this signal to be ANDed with the active region to only display the borders when the counters were within the bounds I defined.



Border Drawing Logic

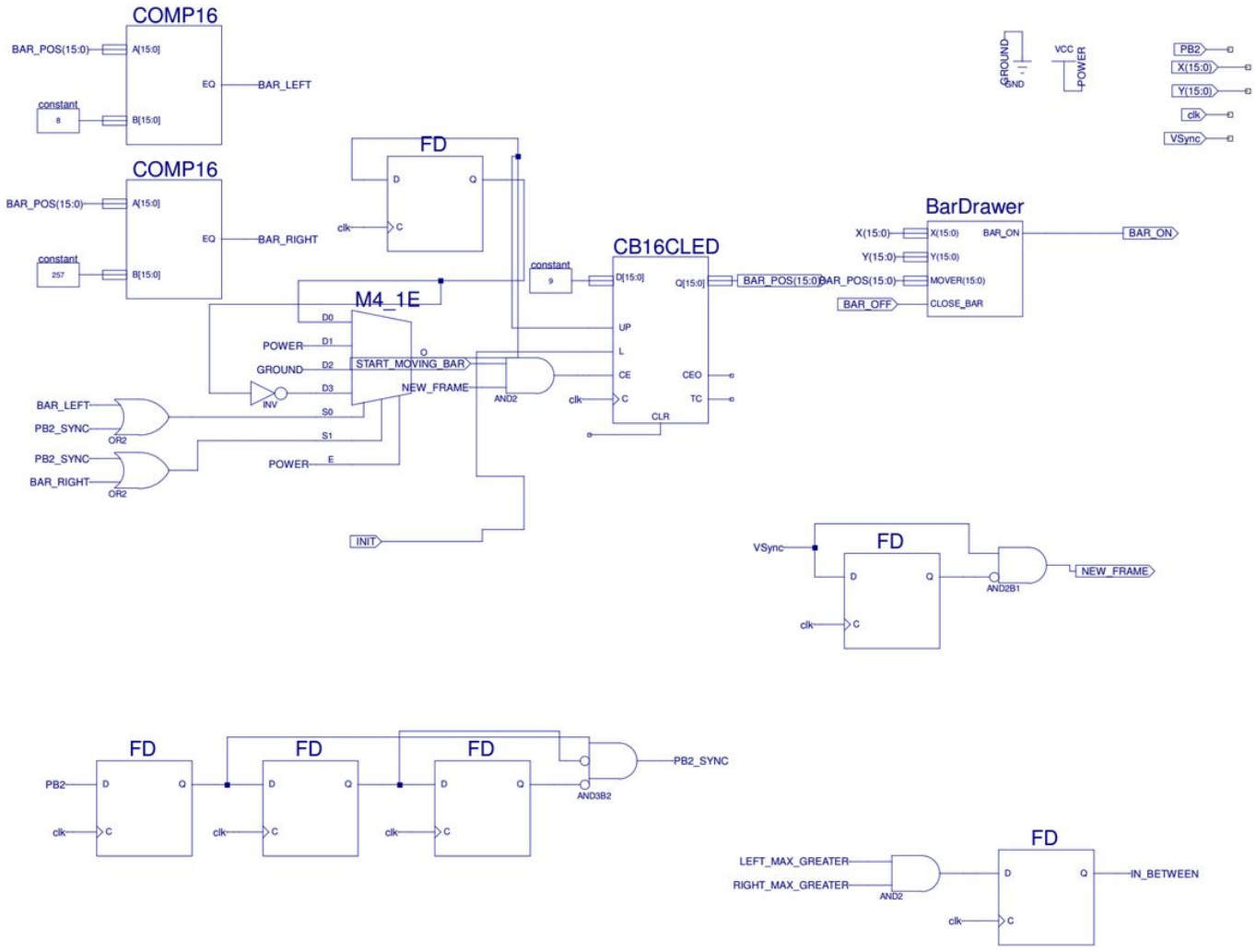
Bar, Gap, & Fly

I used a similar method to draw the moving 4 pixel wide bar horizontally through the center. This bar also has a 32 pixel gap in it that opened and closed on command. I accomplish the gap by drawing the bar in the regions void of the gap. The gap also had to move back and forth, so I used an up/down counter to keep track of the current gap position and added that value to the void area of the bar. The gap had to move in the opposite direction once it hit a border, and this was accomplished by toggling the UP input of that counter to count in the

opposite direction using MUXing logic. The gap also had to change direction with button 2 was pressed.

```
module BarDrawer(
input [15:0] X,
input [15:0] Y,
input [15:0] MOVER,
input CLOSE_BAR,
output BAR_ON
);
assign BAR_ON = ((Y > 399) & (Y < 404) & ~((X > MOVER)&~CLOSE_BAR & (X < (MOVER +
             )&~CLOSE_BAR)));
 endmodule
```

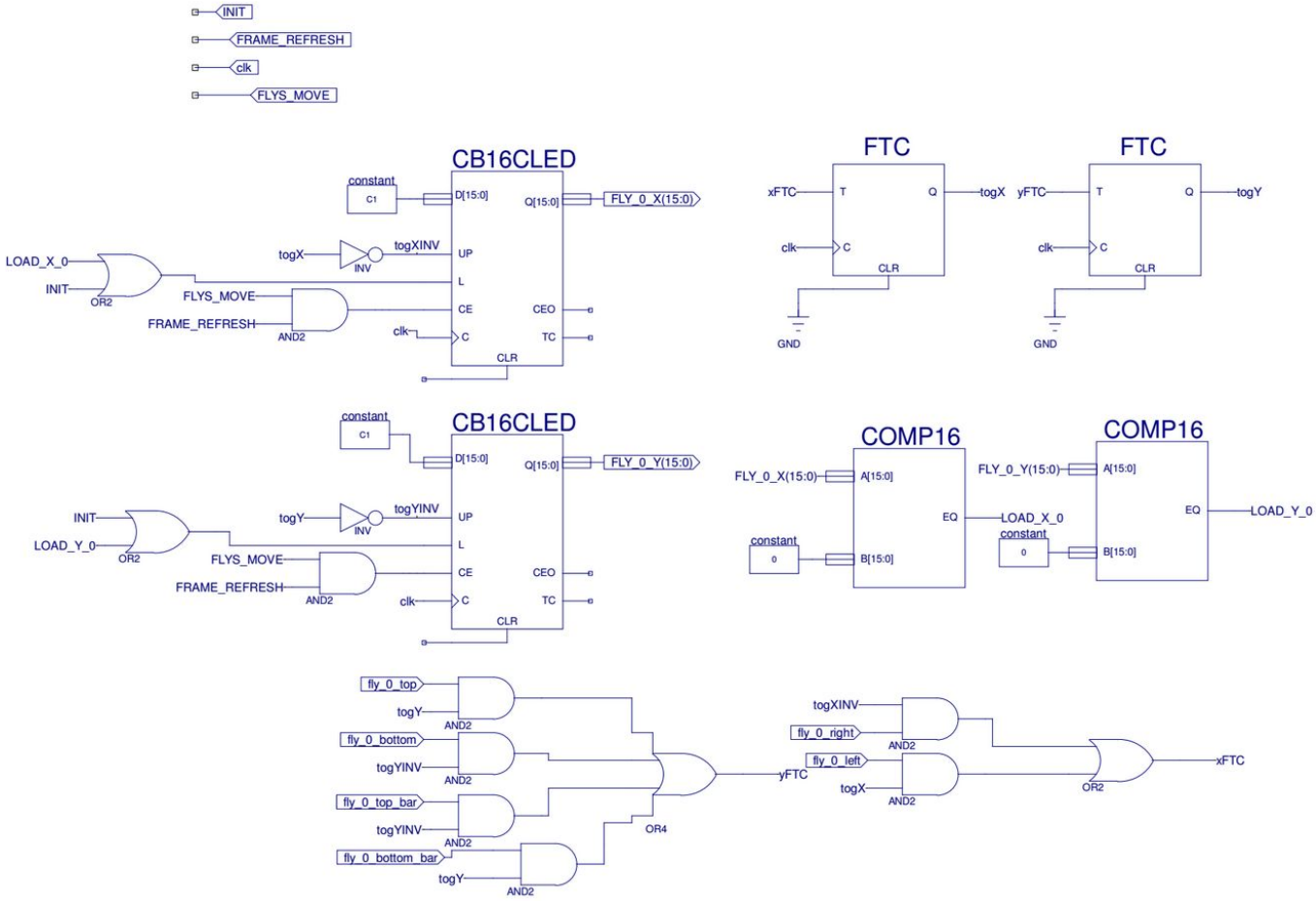
The Mover input was the output of the counter holding the gap position. This module took in push button 2 (PB2) as input, and since that was unjustified human error-ridden input, I had to edge detect the input for PB2 to get clean synchronized input. This was accomplished with 3-D-Flip-Flop edge detector. Several input initialize and manipulate the up/down counter controlling the bar gap, including output from the state machine and an inner mechanism that detects when the gap has hit an edge.


COMP16
BAR_POS(15:0)
A[15:0]
constant
8
B[15:0]
EQ
BAR_LEFT
COMP16
BAR_POS(15:0)
A[15:0]
constant
257
B[15:0]
EQ
BAR_RIGHT
FD
D
Q
clk
C
CB16CLED
constant
9
D[15:0]
Q[15:0]
BAR_POS(15:0)
UP
L
CE
CEO
TC
C
clk
CLR
BarDrawer
X(15:0)
X(15:0)
BAR_ON
BAR_ON
Y(15:0)
Y(15:0)
BAR_POS(15:0)
MOVER(15:0)
BAR_OFF
CLOSE_BAR
M4_1E
D0
POWER
D1
GROUND
D2
START_MOVING_BAR
D3
NEW_FRAME
O
INV
S0
S1
E
AND2
BAR_LEFT
PB2_SYNC
OR2
PB2_SYNC
BAR_RIGHT
OR2
POWER
INIT
GROUND
GND
VCC
POWER
PB2
X(15:0)
Y(15:0)
clk
VSync
VSync
FD
D
Q
clk
C
AND2B1
NEW_FRAME
FD
FD
FD
PB2
D
Q
clk
C
D
Q
clk
C
D
Q
clk
C
AND3B2
PB2_SYNC
FD
LEFT_MAX_GREATER
RIGHT_MAX_GREATER
AND2
D
Q
clk
C
IN_BETWEEN

It was then time to draw the 8 X 8 pixel flies. This task (drawing and moving the flies) was very similar to the above methods to draw and move drawn objects. The flies all had initial positions that I loaded into their X and Y position counters on command.

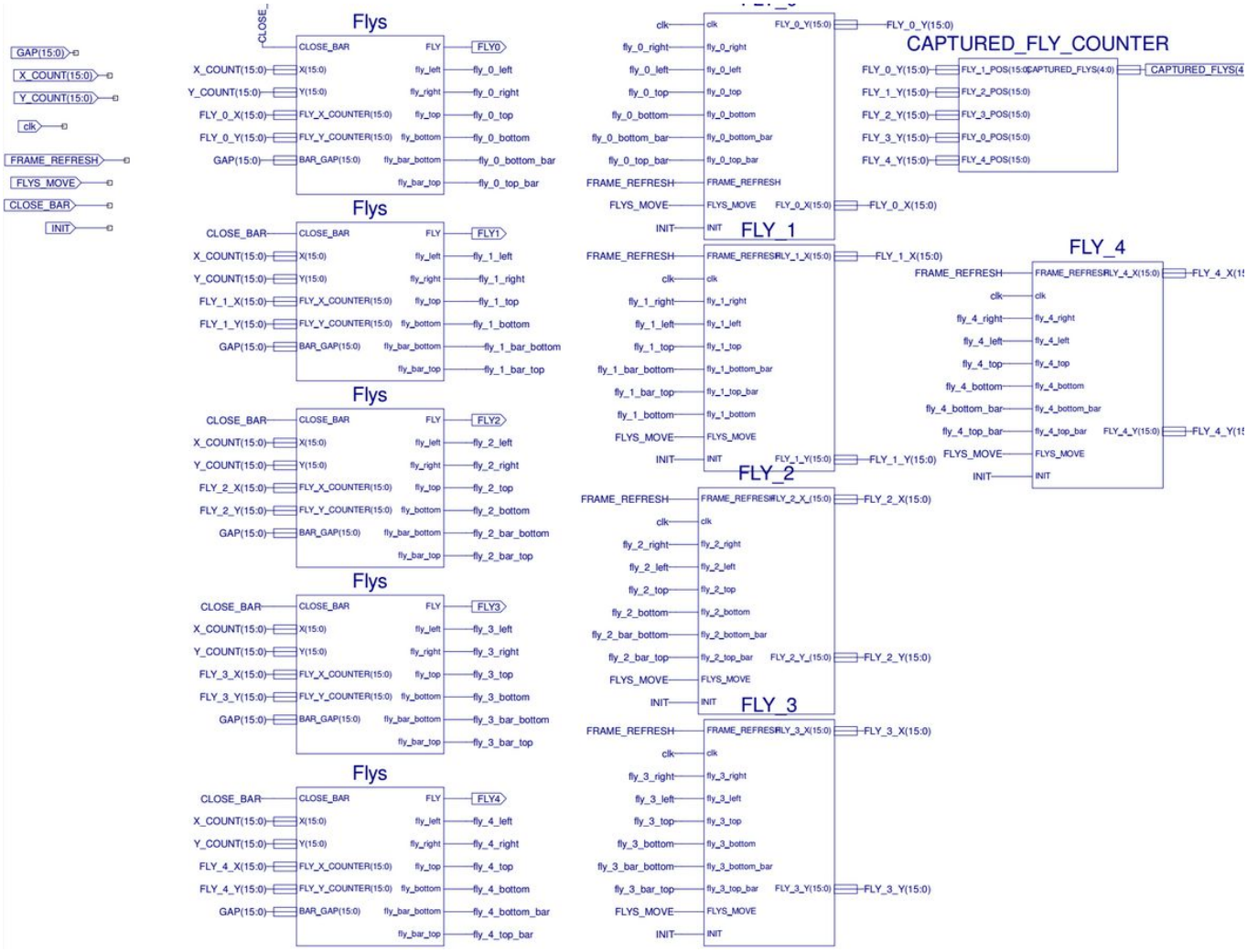
```
module Flys(
    input [15:0] X,
    input [15:0] Y,
    input [15:0] FLY_X_COUNTER,
    input [15:0] FLY_Y_COUNTER,
    input [15:0] BAR_GAP,
    input CLOSE_BAR,
    output FLY,
    output fly_left,
    output fly_right,
    output fly_top,
    output fly_bottom,
    output fly_bar_bottom,
    output fly_bar_top
    );


assign fly_left = FLY_X_COUNTER == 8;
assign fly_right = FLY_X_COUNTER == 624;
assign fly_top = FLY_Y_COUNTER == 8;
assign fly_bottom = FLY_Y_COUNTER == 462;
assign fly_bar_top = (FLY_Y_COUNTER + 8 == 400) & ((FLY_X_COUNTER <= BAR_GAP |
FLY_X_COUNTER >= BAR_GAP + 32)&~CLOSE_BAR) | CLOSE_BAR&(FLY_Y_COUNTER + 8 == 400);
assign fly_bar_bottom = (FLY_Y_COUNTER + 8 == 412) & ((FLY_X_COUNTER <= BAR_GAP |
FLY_X_COUNTER >= BAR_GAP + 32)&~CLOSE_BAR) | CLOSE_BAR&(FLY_Y_COUNTER + 8 == 412);

   assign FLY = (X >= FLY_X_COUNTER) & (X <= FLY_X_COUNTER + 8) & (Y >= FLY_Y_COUNTER)
 & (Y <= FLY_Y_COUNTER + 8); //draws fly

   endmodule
```

Once again, similarly to the bar gap, the fly positions (X and Y) were monitored by two counters and other logic:



Fly Control Logic For A Single Fly. Initial Fly Positions Are Constants Loaded Into The Counters

Since there were five total flies in the game, I repeated the fly controller and logic four more times to form a massive fly colony of control and feedback.

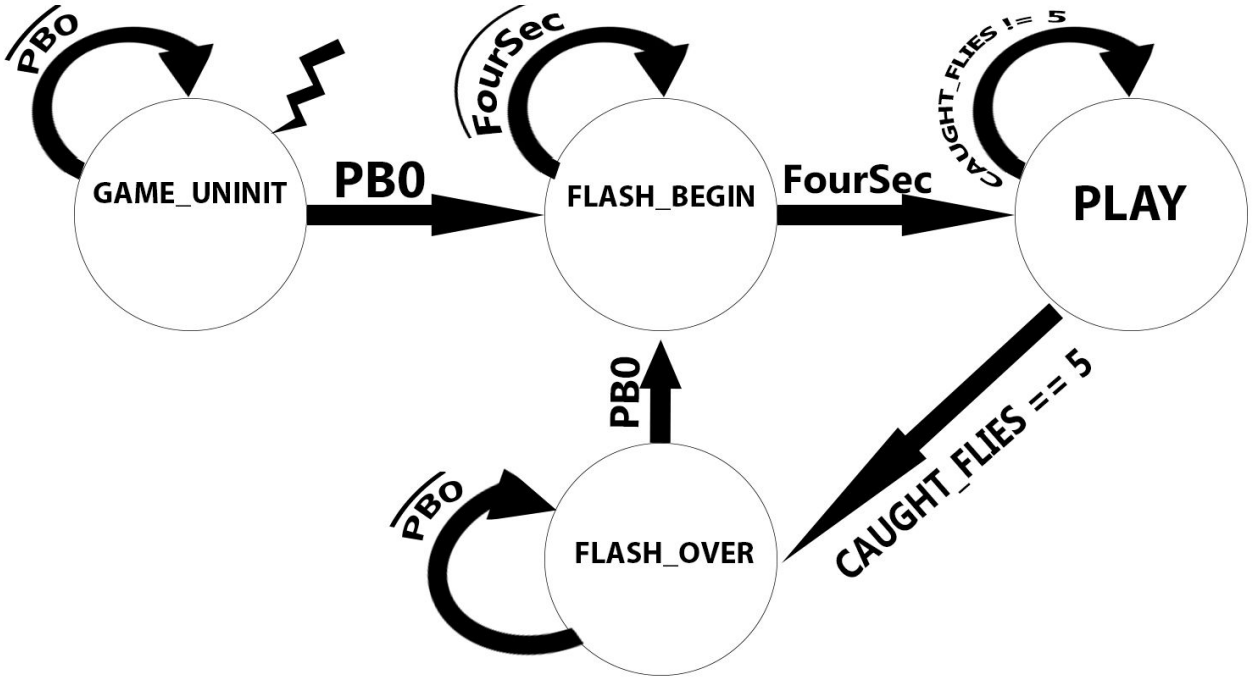


The flies all initially traveled in the same direction (i.e. the counters counted up initially), and then changed one component position on border and bar contact. The flies also had to be able to fly through the gap in the horizontal bar.

The Finite State Machine & Game Controll

The final component of this project was the finite state machine that controlled the game initialization, the fly movement, the presence and absence of the bar gap, and the flashing of the flies upon initialization and when the game was completed.

I used the following state machine to control this game:



INPUTS:

PB0 - User input, push button 0. Initializes the game and resets once complete

FourSec - output from a counter on the top level that counts to four seconds

CAUGHT_FLIES - counts number of currently caught flies (method described below)

OUTPUTS:

CLOSE_BAR - High when the bar should be closed and flies should not move through a gap. Beginning of the game and when all flies are caught.

FLASH_FLY - High when flies should be flashing upon initialization and when all flies caught.

FLY_MOVES - Low when no flies should be moving before initialization.

RESET_POS - Sets the fly position to their initial position when high. Pre-initialization and when game is reset after winning.

STATES:

GAME_UNINIT - Game is uninitialized in this state, and push button 0 is awaited in order to go to the next state. Flies are in their initial position in this state and the bar is closed and not moving.

FLASH_BEGIN - The flies flash for four seconds upon initialization. The gap is opened, but the flies are only flashing, not moving. They are still in their initial positions in this state.

PLAY - The game has been initialized and the user is now playing the game in this state. The machine leaves the PLAY state when the 5-bit vector named, "FLIES_CAUGHT" evaluates to the decimal value 5. This happens when all five flies have been caught and are below the bar.

FLASH_OVER - All five flies have been caught and the bar closes. The flies remain moving and flashing. Push button 0 will reinitialize the game in the FLASH_BEGIN state.

Fly Sensing and 7-Segment Display Logic

In order to know when the flies were in the caught region, I created a module that checked each individual fly's Y-position counter to see if that fly was under the horizontal bar's Y position. If that fly was over that Y limit, I set that individual fly's "UNDER" bit to a high with a 5-bit vector.

I then used Verilog's ability of counting individual bit values in a bit-vector to store the value caught flies into a decimal holding variable.

```
module CAPTURED_FLY_COUNTER(
    input [15:0] FLY_1_POS,
    output [4:0] CAPTURED_FLYS,
    input [15:0] FLY_2_POS,
    input [15:0] FLY_3_POS,
    input [15:0] FLY_0_POS,
    input [15:0] FLY_4_POS
    );



   wire [4:0] CAPT;
   assign CAPT[0] = (FLY_0_POS + 8 >= 412);
   assign CAPT[1] = (FLY_1_POS + 8 >= 412);
   assign CAPT[2] = (FLY_2_POS + 8 >= 412);
   assign CAPT[3] = (FLY_3_POS + 8 >= 412);
   assign CAPT[4] = (FLY_4_POS + 8 >= 412);


assign CAPTURED_FLYS = CAPT[0] + CAPT[1] + CAPT[2] + CAPT[3] + CAPT[4];


endmodule
```

412 is the Y position of the bottom of the horizontal gap

The FLIES_CAUGHT vector is used within the state machine to know when the game is over, but I also used this knowledge to my advantage for the 7-segment display to display the number of caught flies on the board's 7-segment anode.
Since the anode could only display six possible values (0-5), it was a reasonable decision to hard code what segments within the anode had to be on corresponding to the FLIES_CAUGHT vector.

```
module SEVEN_SEG(
    input [4:0] CAPT_FLYS,
    output CCA,
    output CCB,
    output CCC,
    output CCD,
    output CCE,
    output CCF,
    output CCG
    );



assign CCA = ~((CAPT_FLYS == 2) | (CAPT_FLYS == 3) | (CAPT_FLYS == 5) | (CAPT_FLYS ==
0));
assign CCB = ~((CAPT_FLYS == 1) | (CAPT_FLYS == 2) | (CAPT_FLYS == 3) | (CAPT_FLYS ==
4) | (CAPT_FLYS == 0));
assign CCC = ~((CAPT_FLYS == 1) | (CAPT_FLYS == 3) | (CAPT_FLYS == 4) | (CAPT_FLYS ==
5) | (CAPT_FLYS == 0));
assign CCD = ~((CAPT_FLYS == 2) | (CAPT_FLYS == 3) | (CAPT_FLYS == 5) | (CAPT_FLYS ==
0));
assign CCE = ~((CAPT_FLYS == 2) | (CAPT_FLYS == 0));
assign CCF = ~((CAPT_FLYS == 4) | (CAPT_FLYS == 5) | (CAPT_FLYS == 0));
assign CCG = ~((CAPT_FLYS == 2) | (CAPT_FLYS == 3) | (CAPT_FLYS == 4) | (CAPT_FLYS ==
5));


endmodule
```

Hard-Coded 7-Segmentation Logic

VGA Color Output & Flashing

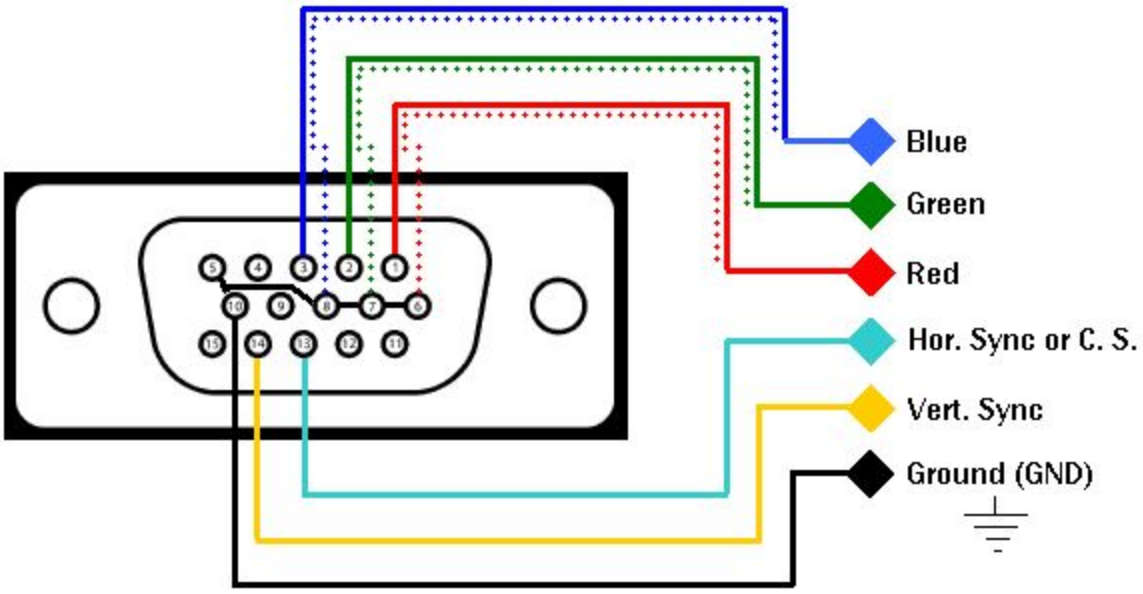


Photo Courtesy of StackExchange

Besides using the correct pins for the H and V Sync, the VGA technology allows 8-bit RGB values to be outputted to the monitor via pins 1, 2, and 3 in the following vector.

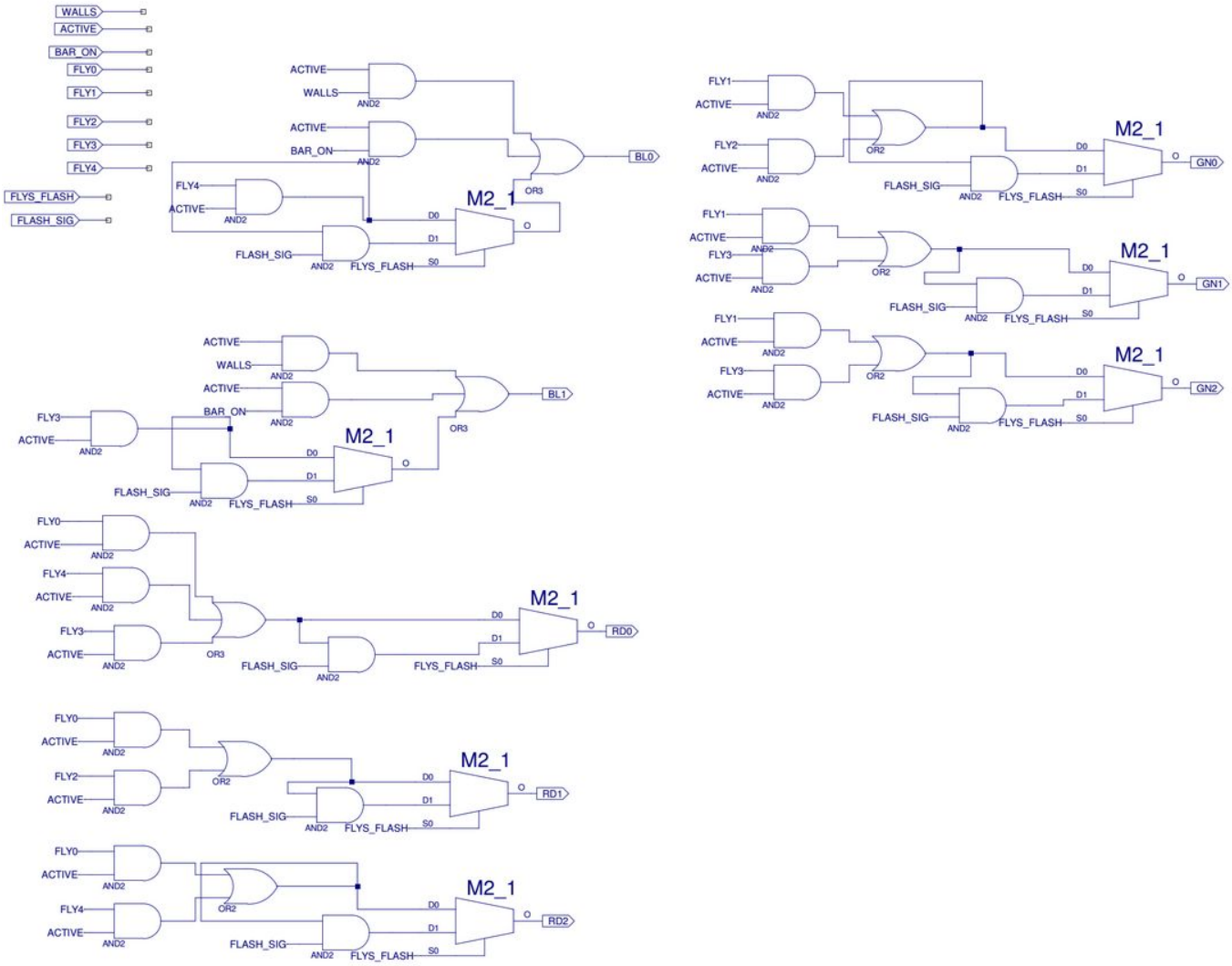
RED2 RED1 RED0 GREEN2 GREEN1 GREEN0 BLUE1 BLUE0

255 possible colors could be displayed through an 8-bit color scheme, and that is how I accomplished different fly colors.

In order to make a certain pixel a certain color, I logically ANDed the pixel I wanted to turn on, whether it was a fly in motion or the moving bar, with the active region to be outputted to the pin of the VGA corresponding to the colors I wanted.

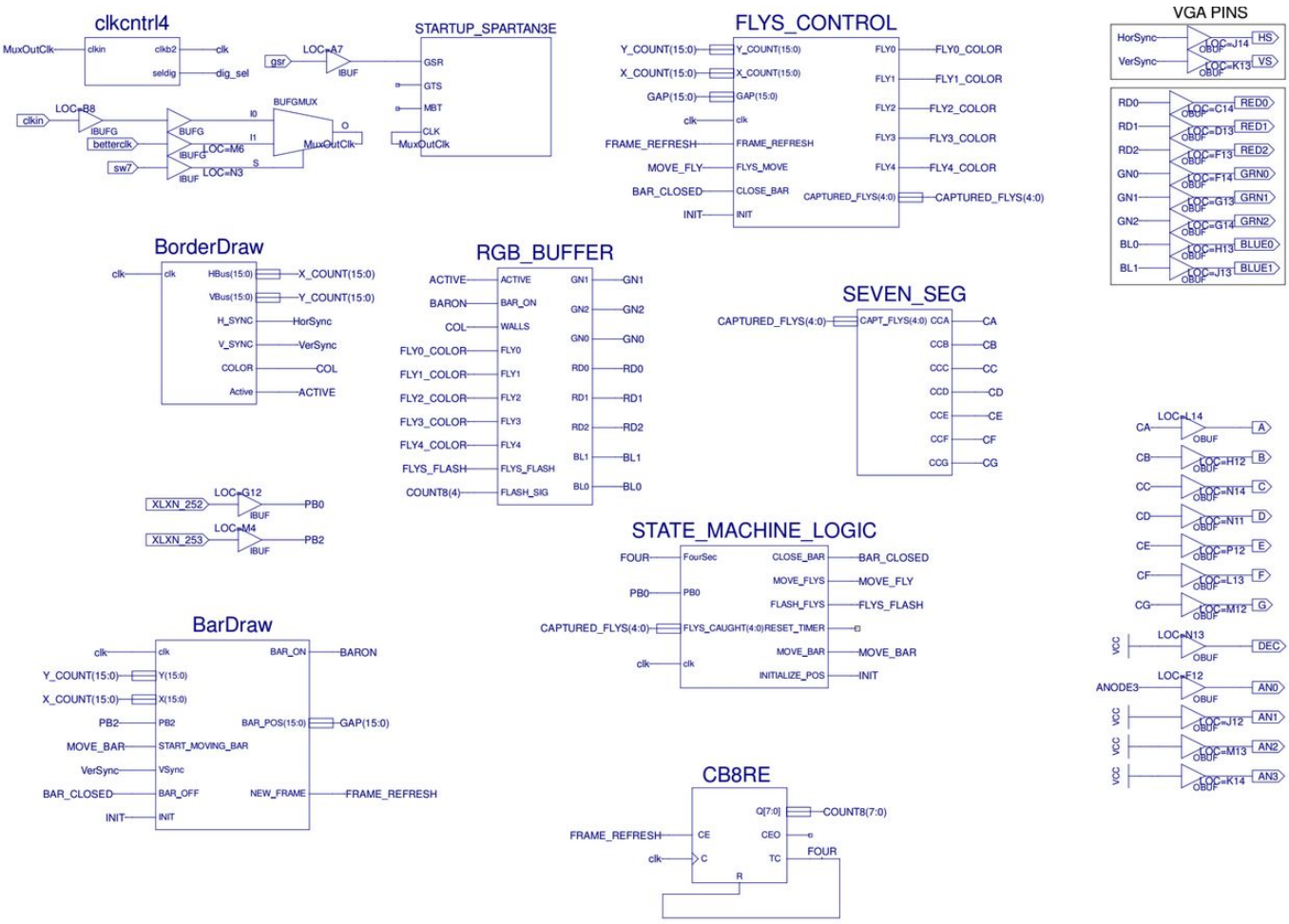
I created a RGB Buffer module that handled this task by taking in my incoming signals and outputting a specific color to the specific coordinate to write to.

This symbol also took the job of flashing the flies. The FLASH_FLIES coming from the state machine and a FLASH_SIG symbol that oscillated on and off and a human legible pace, flash the flies at the FLASH_SIG frequency when FLASH_FLIES was high.



RGB Buffer That Handles Flashing & Color Output

Top Level Module



The top level is where I tied all of the components and modules I created together and set the pins to their locations on the board. The 8-bit counter at the bottom counted to four seconds and it also provided me a steady oscillation for flashing via the most significant bit of the output.

PINS USED

BUTTONS: BTN0: G12 - BTN2: M4

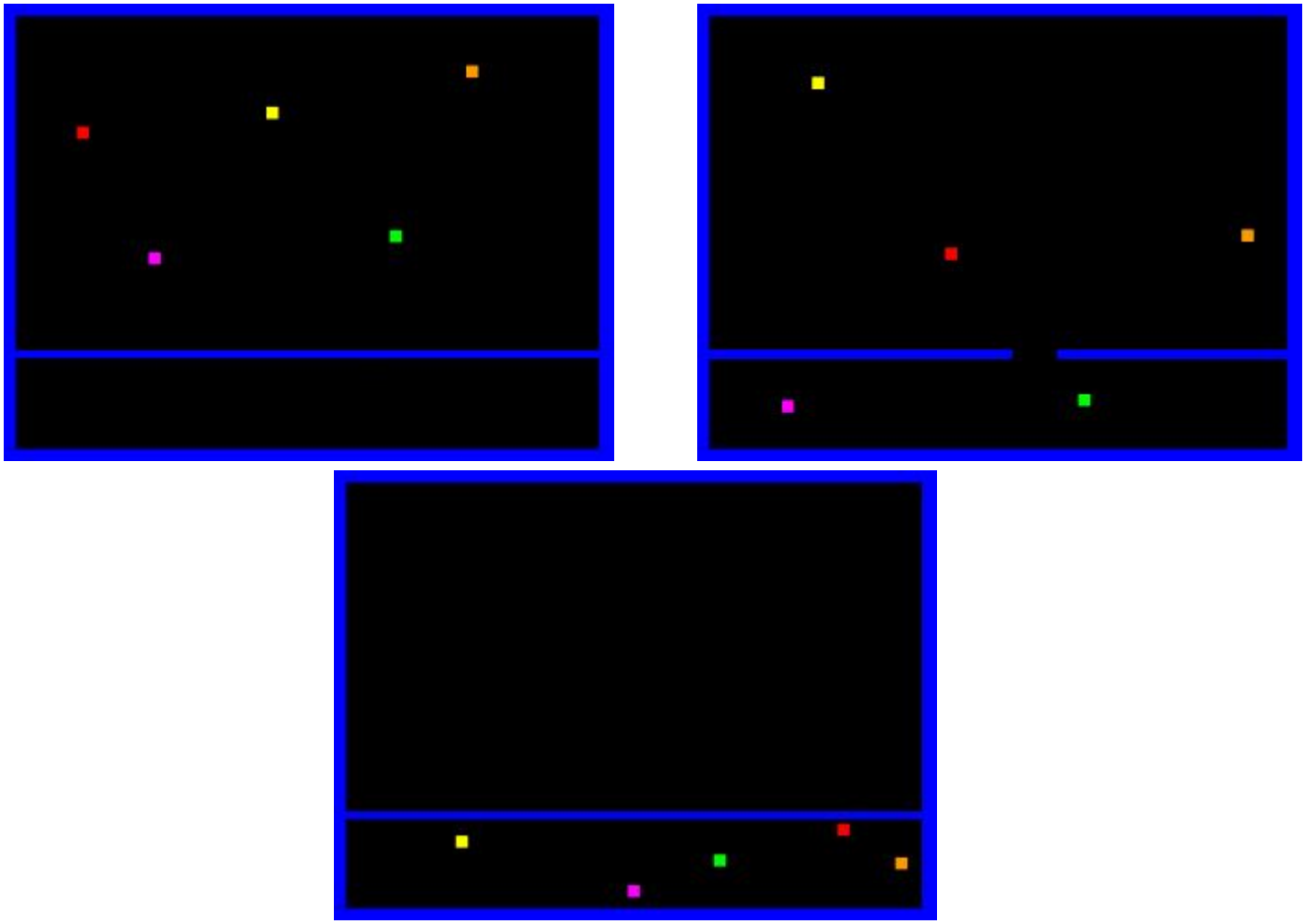
VGA PINS: HS: J14 - VS: K13 - RED0: C14 - RED1: D13 - RED2: F13 - GRN0: F14 - GRN1: G13
GRN2: G14 - BLUE0: H13 - BLUE1: J13

ANODES & SEGMENTS: DP: N13 - AN0: F12 - AN1: J12 - AN2: M13 - AN3: K14 - CA: L14
CB: H12 - CC: N14 - CD: N11 - CE: P12 - CF: L13 - CG: M12

CLOCKS: CRYSTAL: M6 - ELECTRONIC CLOCK: B8

SWITCHES: SWITCH 7 (FOR SWITCHING CLOCKS): N3

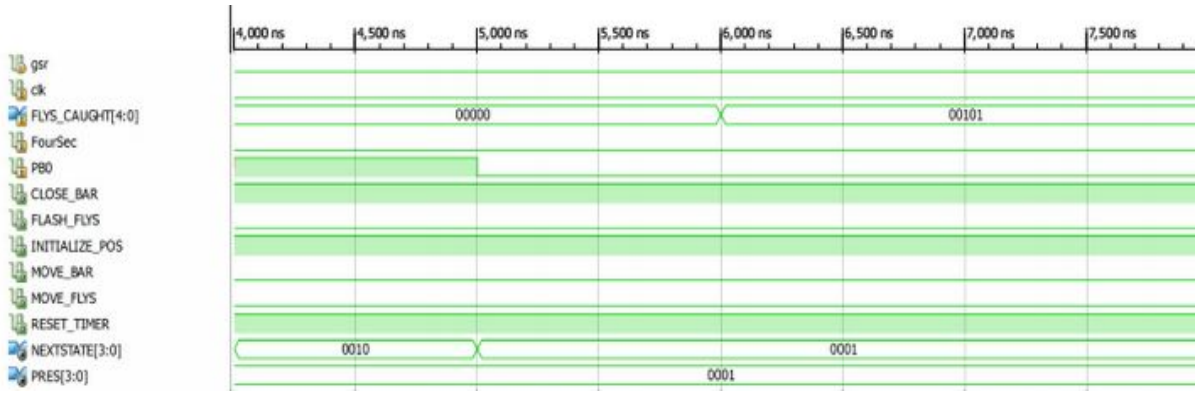
Results



The three screenshots above show the game in its three states. Uninitialized, play mode, and over.

The rightmost anode displays the number of captured flies. The game is over at 5.

The state machine was simulated and when certain states were forced, the correct transition took place.



The FLYS_CAUGHT vector < 5, so the game is still in the PLAY state

Additional Question

Maximum Clock Speed:

According to my static timing analysis report, the longest path from the clock to any pin, was 19.101 nanoseconds, so therefore, my minimum clock frequency was:

$$f_{MIN} = \frac{1}{19.101 \times 10^{-9}\ seconds} = 52.353\ MHz$$

Maximum Clock Frequency:

$$f_{MAX} = \frac{1}{13.585 \times 10^{-9} seconds} = 73.556\ MHz$$

Based off of my timing analysis:

Clock betterclk to Pad

Destination	clk (edge) to PAD	Internal Clock(s)	Clock Phase
A	16.436(R)	clk	0.000
B	14.280(R)	clk	0.000
BLUE0	15.931(R)	clk	0.000
BLUE1	19.101(R)	clk	0.000
C	15.872(R)	clk	0.000
D	15.965(R)	clk	0.000
E	15.264(R)	clk	0.000
F	14.921(R)	clk	0.000
G	14.468(R)	clk	0.000
GRN0	16.817(R)	clk	0.000
GRN1	16.465(R)	clk	0.000
GRN2	15.997(R)	clk	0.000
HS	16.538(R)	clk	0.000
RED0	16.832(R)	clk	0.000
RED1	16.997(R)	clk	0.000
RED2	17.860(R)	clk	0.000
VS	11.754(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	13.595			