University of California Santa Cruz
Department of Computer Engineering
Lab Experiment Report # 7
Floppy Slug

Author: Kyle Jeffrey
Lab Partner: NA
Due Date: 3/15/19

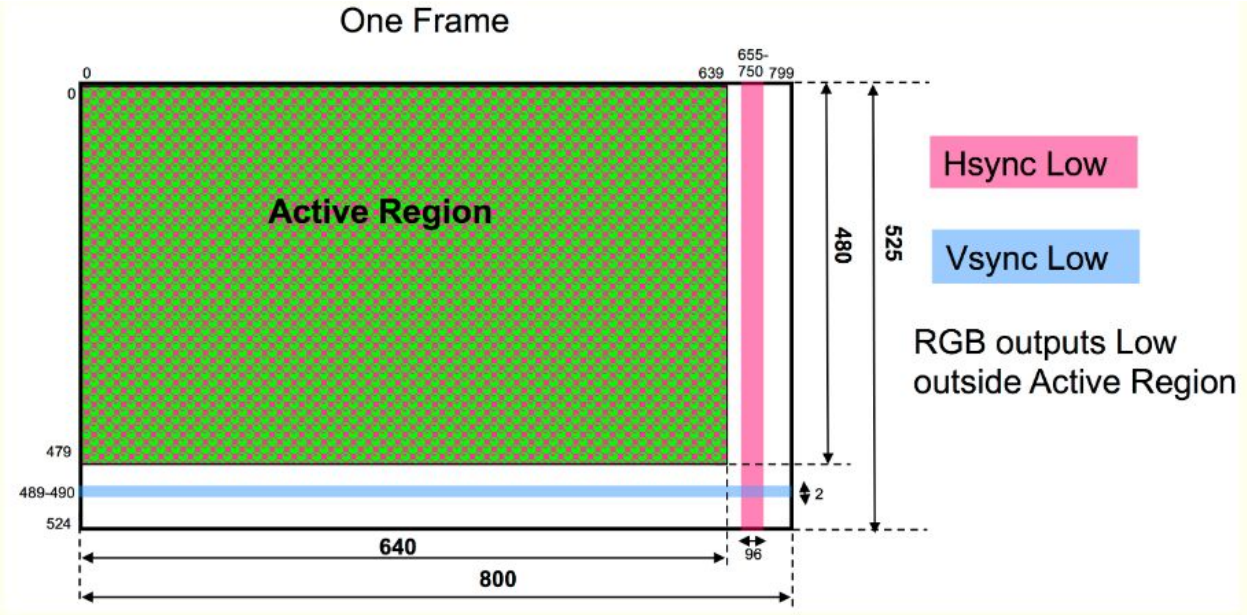
Objective

Using the Basys 3 board design the game “Floppy Slug” to the designated parameters provided by the lab instructions, using the VGA port on the board to display to an LCD screen.

Kyle Jeffrey
Prof Schlag
CE 100
2/21/18
2

VGA Controller:

The VGA controller manages the data signals sent to through the VGA port to an LCD screen. The LCD screen needs to be given an HSync, a VSync, and 12 RGB bits determining the color being printed to each pixel. The RGB signals were controlled on the top level module for ease of programming. Following the convention of old CRT screens, the monitor prints like a typewriter would. From left to right, printing pixel by pixel, until it hits the edge of the screen and then resets back to the left side, one row down. This occurs until you hit the bottom of the screen and the monitor resets to the top again.

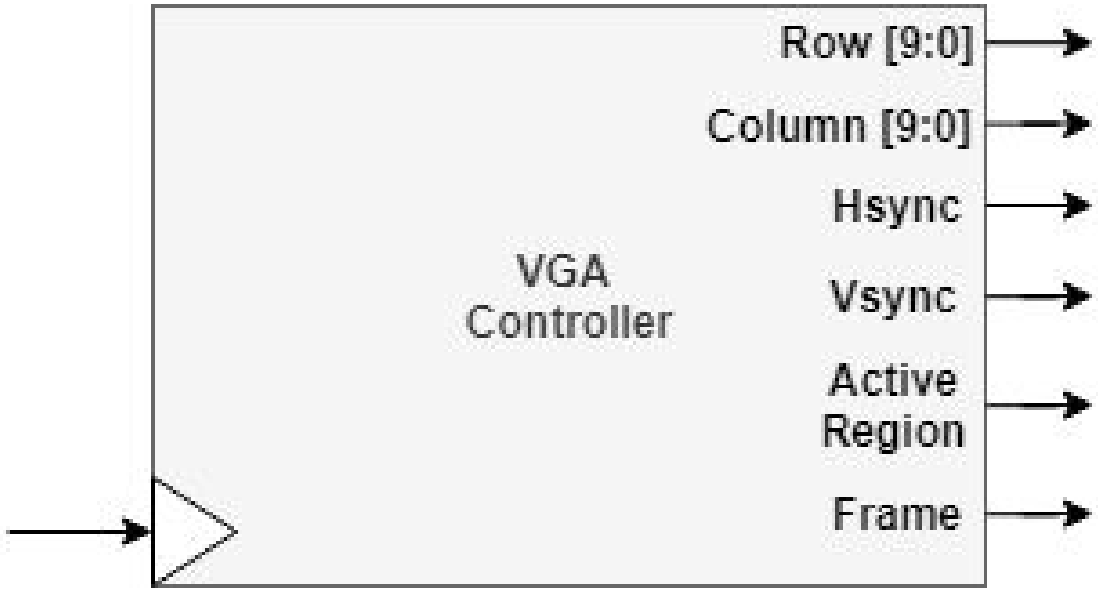


Taken from CE100 Course website

Method:

Hsync and Vsync needed to be high all places but within a small range. Hsync is low from columns 655-750, and Vsync low between 489-490. The controller had two counters that kept track of the columns and rows. After the column counter hits it's max value, 799, the row counter incremented by 1, and the column counter would reset to 0. This continues until the row counter reaches max value, 524, at which point the row counter would reset to 0. A signal for Active Region and Frame was included to help with printing colors to the screen on the top level, and keeping track of time.

Results:



<u>Simulation:</u> The counter needed to meet a very specific set of requirements or the LCD screen would not start up. The columns had to count to exactly 799 and the rows had to count to exactly 524. This was one of the very few stages in the lab that a simulation could be done to test functionality. The VGA controller took just a clock input and needed to count columns from 0-799 and increase 1 in rows after the column counter hit 799.

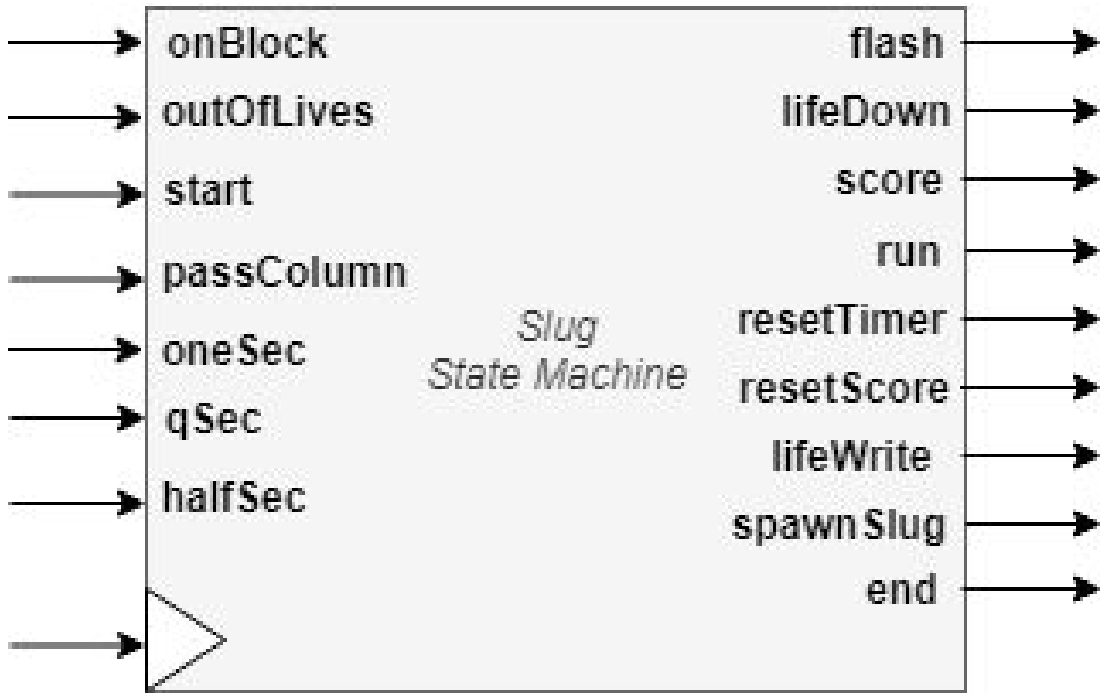
Slug State Machine:

This state machine controlled the general game flow logic. This state machine didn't include signals for the obstacle spawning to prevent convulsion.

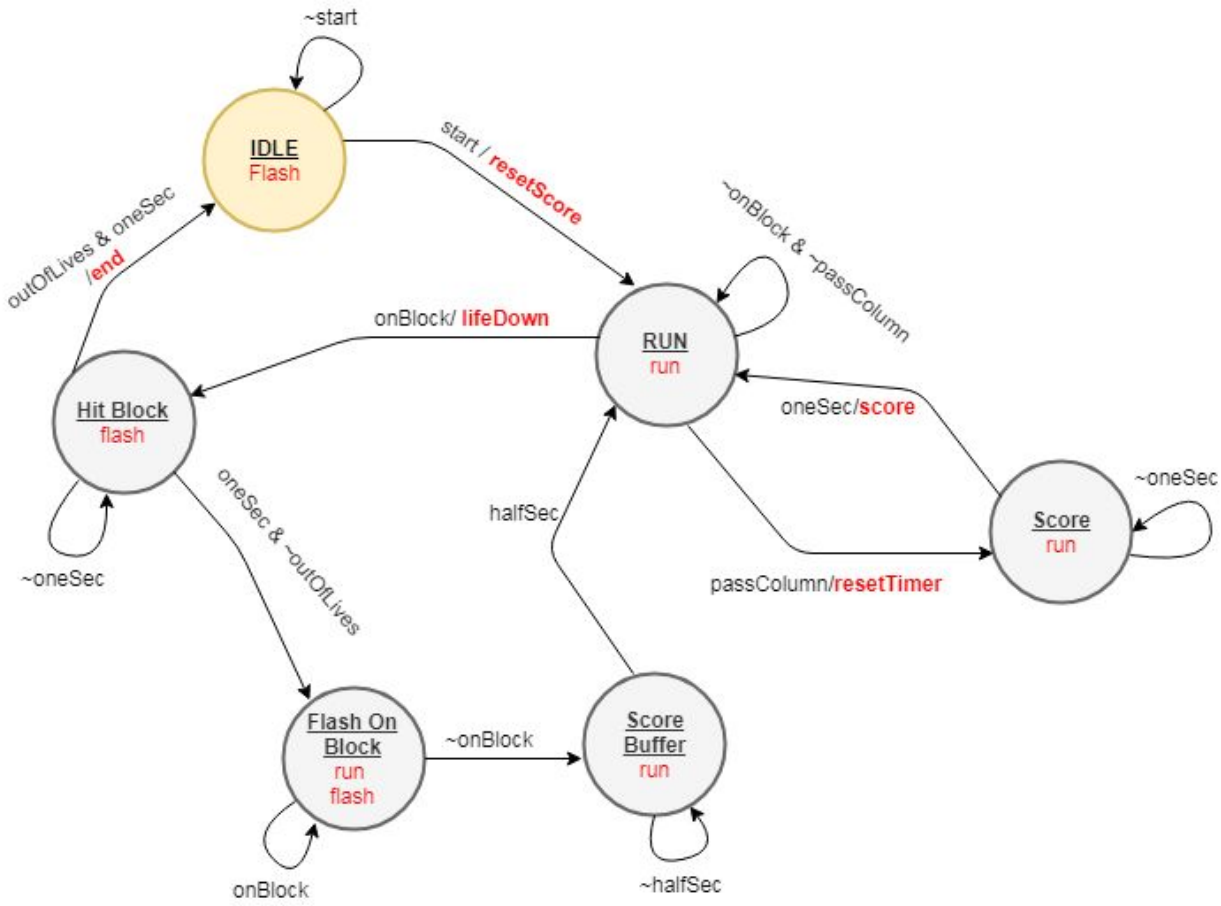
Kyle Jeffrey
Prof Schlag
CE 100
2/21/18

Method:
A state diagram was drawn and next state logic was derived using one-hot encoding. The slug state machine controlled stopping the game when hitting an object, but another state machine was used for spawning the obstacles. A couple notes on the state diagram: There is a score buffer between hitting a block, staying on the black and flashing, and going back to idle called, “score buffer”. This solves the issue of scoring on an obstacle column that the slug has hit. By waiting a half second, 30 frames, it assures that once the slug hits the obstacle which is 24 frames wide, and is about 50 frames away from the next column, the slug can hit the Run state before the next column, and that it will assure there’s no way of scoring on the same column the slug has hit. The “score” state also had a buffer of 1 second to prevent any double or greater scoring on the same column. The input signal “passColumn” could be high for several clock cycles, so we wanted to avoid scoring more than one point.

Results:



Kyle Jeffrey
Prof Schlag
CE 100
2/21/18
5



Inputs:
- onBlock
- outOfLives
- Start
- PassColumn
- OneSec
- qSec
- halfSec

Outputs:
- Flash
- lifeDown
- score
- run
- resetTimer
- resetScore
- lifeWrite
- spawnSlug
- end

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

Next State Logic:

- D_{IDLE} = ($Q_{Hit\ Block}$ & oneSec & outOfLives) | (Q_{IDLE} & ~start)
- D_{Run} = (Q_{IDLE} & start) | (Q_{Run} & ~onBlock & ~passColumn) | (Q_{Score} & qsec) | ($Q_{Score\ Buffer}$ & halfSec)

- $D_{Hit\ Block}$ = (Q_{Run} & onBlock) | ($Q_{Hit\ Block}$ & ~oneSec)
- $D_{Flash\ on\ Block}$ = ($Q_{Hit\ Block}$ & oneSec & ~outOfLives) | ($Q_{Flash\ On\ Block}$ & onBlock);
- D_{Score} = (Q_{IDLE} & passColumn) | (Q_{Score} & ~qsec)
- $D_{Score\ Buffer}$ = ($Q_{Flash\ On\ Block}$ & ~onBlock) | ($Q_{Score\ Buffer}$ & ~halfSec)

Outputs:

- flash = Q_{IDLE} | Q_{IDLE} | Q_{IDLE}
- lifeDown = Q_{Run} & onBlock
- scoreUp = Q_{Score} & qsec
- Run = Q_{Run} | $Q_{Flash\ on\ Block}$ | Q_{Score} | $Q_{Score\ Buffer}$
- resetTimer = (Q_{Run} & onBlock) | (Q_{Run} & passColumn) | ($Q_{Flash\ on\ Block}$ & ~onBlock)
- resetScore = Q_{IDLE} & start
- spawnSlug = Q_{IDLE}
- End = $Q_{Hit\ Block}$ & oneSec & outOfLives

<u>Simulation:</u> Going from state to state needed to flow correctly. I did originally hit some bugs were the logic would jump into multiple states at the same time because the next state logic didn't account for more specific inputs.

Kyle Jeffrey
Prof Schlag
CE 100
2/21/18
7

Obstacle State Machine:

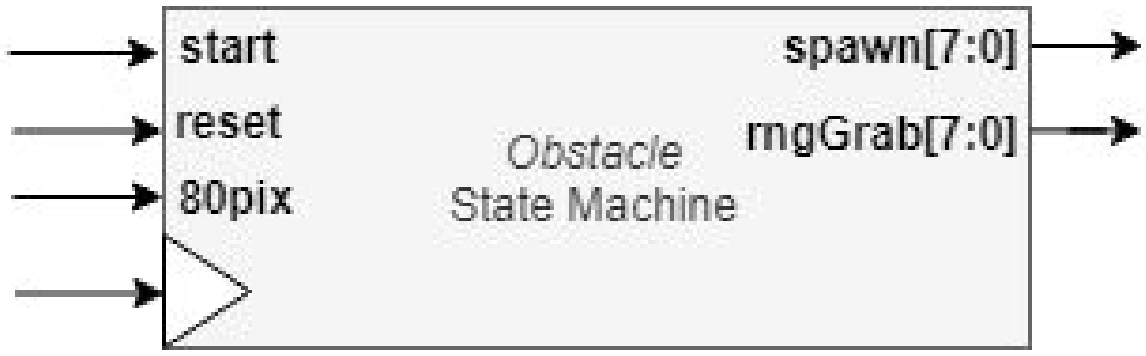
This machine controlled spawning the obstacle for the first time. The state machine only goes through once during every play through to separate the obstacle columns the correct amount of pixels away from each other.

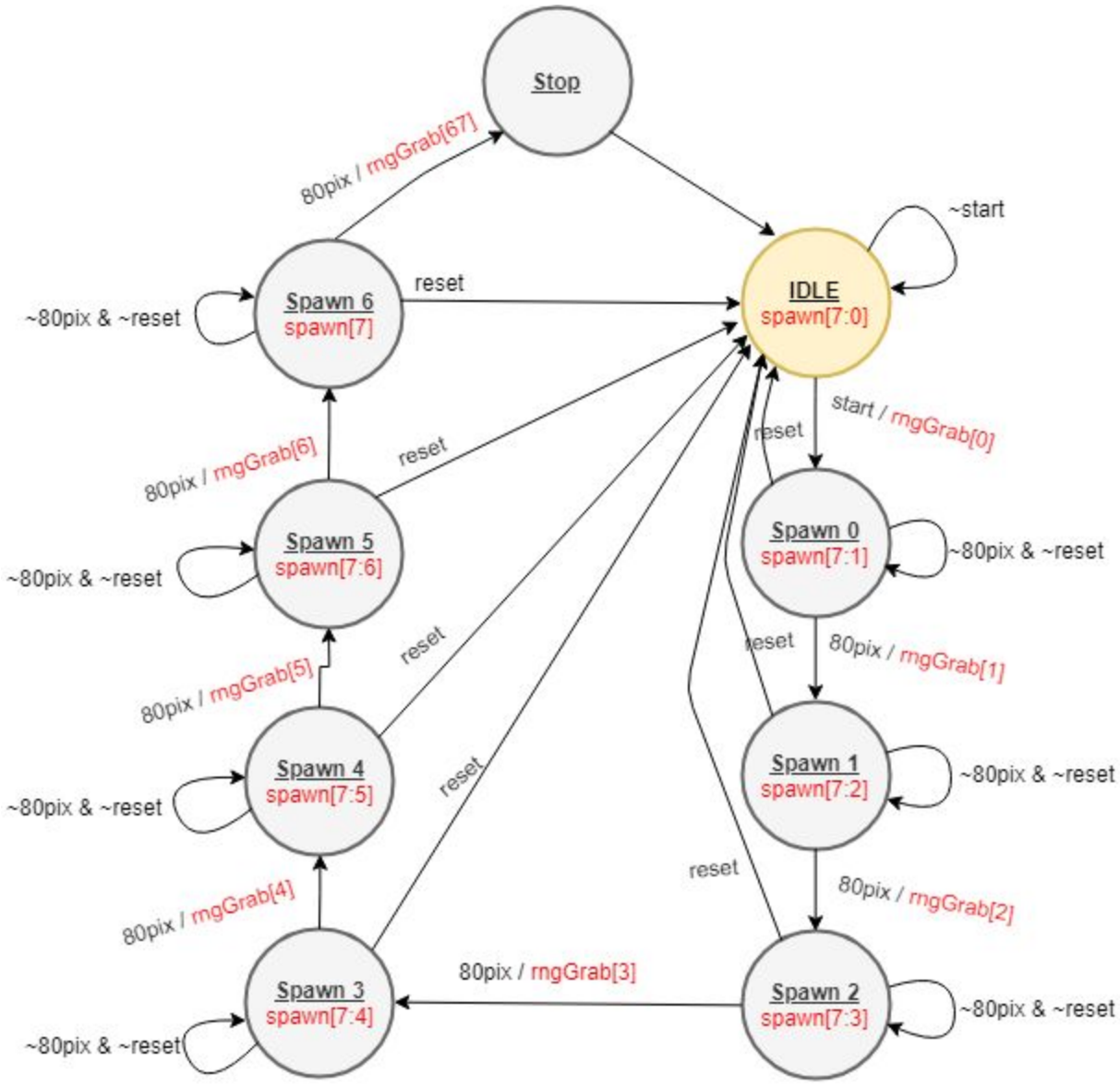
Method:

A state diagram was drawn and next state logic was derived using one-hot encoding. It was paramount that the obstacle spawner had the exact time when to spawn the obstacle columns so the columns were separated correctly. They had to spawn every 80 pixels, and the columns moved a pixel a frame to the left, so this meant that every 80 frames a column needed to be spawned.

This state machine was one of the last modules created as it was necessary to figure out the parameters for 1 obstacle column and how hit detection would work before replicating it. The diagram can look confusing because of the 8 bus outputs. The state machine is based of how the obstacle columns were spawned. Each column had a counter that would decrease to move it left across the screen every frame while the game was running. The load control of each column counter was the output of the state machine “spawn[7:0]” respectively, and the load data was a hardcoded value just off the screen to the right for all the columns. When spawn[0] went low, then the first column would no longer have a load control overriding the countdown of the column counter. This then happened for each subsequent column until all spawned on the screen. The counter’s themselves would then handle the columns resetting.

Results:





Inputs:
- start
- reset
- 80pix

Outputs:
- spawn[7:0]
- rngGrab[7:0]

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

Next State Logic:

- $D_{IDLE} = Q_{IDLE}$ & ~start | (reset)
- $D_{spawn0} = (Q_{IDLE}$ & start | Q_{spawn0} & ~eightyPixels) & ~reset
- $D_{spawn1} = (Q_{spawn0}$ & eightyPixels | Q_{spawn1} & ~eightyPixels) & ~reset
- $D_{spawn2} = (Q_{spawn1}$ & eightyPixels | Q_{spawn2} & ~eightyPixels) & ~reset
- $D_{spawn3} = (Q_{spawn2}$ & eightyPixels | Q_{spawn3} & ~eightyPixels) & ~reset
- $D_{spawn4} = (Q_{spawn3}$ & eightyPixels | Q_{spawn4} & ~eightyPixels) & ~reset
- $D_{spawn5} = (Q_{spawn4}$ & eightyPixels | Q_{spawn5} & ~eightyPixels) & ~reset
- $D_{spawn6} = (Q_{spawn5}$ & eightyPixels | Q_{spawn6} & ~eightyPixels) & ~reset
- $D_{stop} = (Q_{spawn6}$ & eightyPixels | (Q_{stop} & (~eightyPixels | eightyPixels))) & ~reset

Outputs:

- spawn[0] = Q_{IDLE}
- spawn[1] = Q_{IDLE} | Q_{spawn0}
- spawn[2] = Q_{IDLE} | $Q_{spawn[1:0]}$
- spawn[3] = Q_{IDLE} | $Q_{spawn[2:0]}$
- spawn[4] = Q_{IDLE} | $Q_{spawn[3:0]}$
- spawn[5] = Q_{IDLE} | $Q_{spawn[4:0]}$
- spawn[6] = Q_{IDLE} | $Q_{spawn[5:0]}$
- spawn[7] = Q_{IDLE} | $Q_{spawn[6:0]}$
- rngGrab[0] = Q_{IDLE} & start
- rngGrab[7:1] = $Q_{spawn[7:1]}$ & {8{eightyPixels}}

Simulation: The states needed to jump correctly without jumping into two at the same time.

Kyle Jeffrey
Prof Schlag
CE 100
2/21/18
10

Counters:

All of the counters more or less had the same structure with minor tweaks to account for differences in needed behavior.

Lives Counter: This counter tracked how many lives the player had. Only the lowest 4 bits were used which were loaded with the switches. Load control which would input the value was controlled by the slug state machine with the signal “lifewrite”.

Score Counter: This counter tracked the score. The reset was tied to “resetScore” coming from the slug state machine and the up was set to “scoreUp” also coming from the slug state machine.

Time Counter/ Frame Counter: This counter tracked the frames passed, which also served as a way to track the amount of time passed. Up was set to “frame” a signal coming from the vga controller that went high once a frame. Reset was set to “resetTimer” coming from the slug state machine.

Pixel Counter: This counter gave the signal “80pix” signal to the obstacle state machine. By 80 pixels it meant that an obstacle had traveled 80 pixels, meaning 80 frames had passed, as the obstacles moved to the left a pixel a frame.

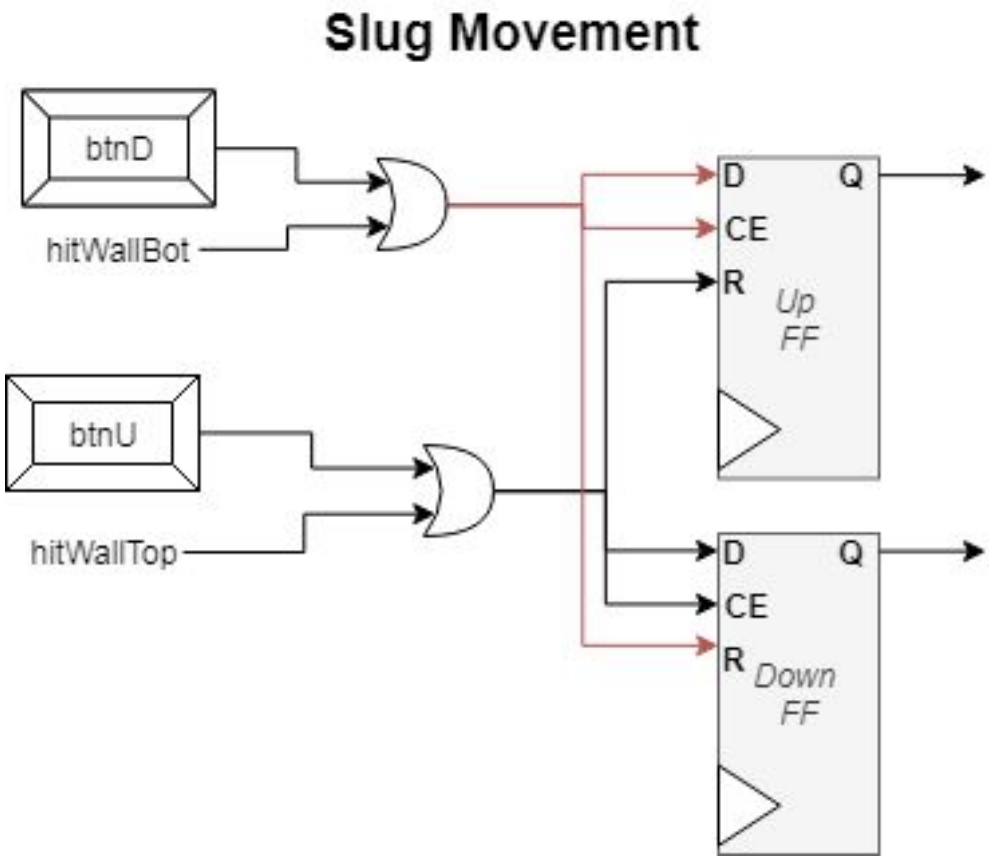
Slug Move Counter: This counter controlled the slug moving up or down. The load control was given by the “spawn slug” signal from the slug state machine. The signal was true during the IDLE state, keeping the slug at its Load Data value which was hard coded to the center pixel of the screen. The up and down signals were given from to flip flops holding the direction the slug should move in.

Method:
All of these counters were minor variations of the 16-bit counter module developed earlier in the academic quarter, therefore designing them was mostly about minor changes throughout the entire process of the game development.

Slug Movement:
The slug movement was handled on the top level, rather than designing a state machine. This logic controlled the slug going up or down, as it didn’t move left to right.

Method:
One flip flop for each direction was used to hold the direction value. These flip flops had the Clock Enable and the Input Data tied to one signal, so that the flip flop would only write a 1 and then hold that value. The resets on the FF’s were tied to the inverse direction. This meant that when the player hit btnU to go up, this signal would write a 1 to the up flip flop and would reset the down flip flop to 0. This also applied to the converse. The outputs then went to the slug movement counter. Hitting the top wall would also reset the up flip flop and set the down flip flop high as well as the converse being true.

Results:
See next page.



Obstacle Column:

The obstacle column was encapsulated in one module and then an array of 8 were instantiated to create 8 obstacle columns. The module took in a bunch of inputs to return a drawing bit for the rgb signals to draw, a hit detection bit to signal when the slug was on one of the obstacles, and a signal for when the slug would pass a column. Each obstacle in the column was offset by 128 pixels from bottom to bottom or top to top.

Method:

Building the obstacles was a slow process that occured in small increments. Originally, all that was made was one obstacle that would hit detect. Then an obstacle column was drawn that

would detect a hit with any of the obstacles. This was then increased to the 8 columns that were drawn on the final game design.

Drawing Bit (Output: "obstacleColumn"): The drawing bit simply gives a 0 or 1 based on column and row coordinates ("x" and "y"). This bit needed to create a column of four blocks with each block 24x32 pixels when sent to the VGA controller to be drawn. This was done with comparator logic. The below verilog code shows how each obstacle was defined.

```
assign obstacle0  = ( y>obstacleBot0  & y<obstacleTop0 )    & (x>obstacleLeft & x<obstacleRight);
assign obstacle1  = ( y>obstacleBot1  & y<obstacleTop1 )    & (x>obstacleLeft & x<obstacleRight);
assign obstacle2  = ( y>obstacleBot2  & y<obstacleTop2 )    & (x>obstacleLeft & x<obstacleRight);
assign obstacle3  = ( y>obstacleBot3  & y<obstacleTop3 )    & (x>obstacleLeft & x<obstacleRight);

assign obstacleColumn  = obstacle0 | obstacle1 | obstacle2 | obstacle3 ;
```

Passing Column (Output: "passColumn"): This signal went high when the left side of the slug passed the right side of the obstacle column, or when the coordinates were equal to each other.

```
assign passColumn = slugLeft == obstacleRight;  //PASS COLUMN FOR SCORE
```

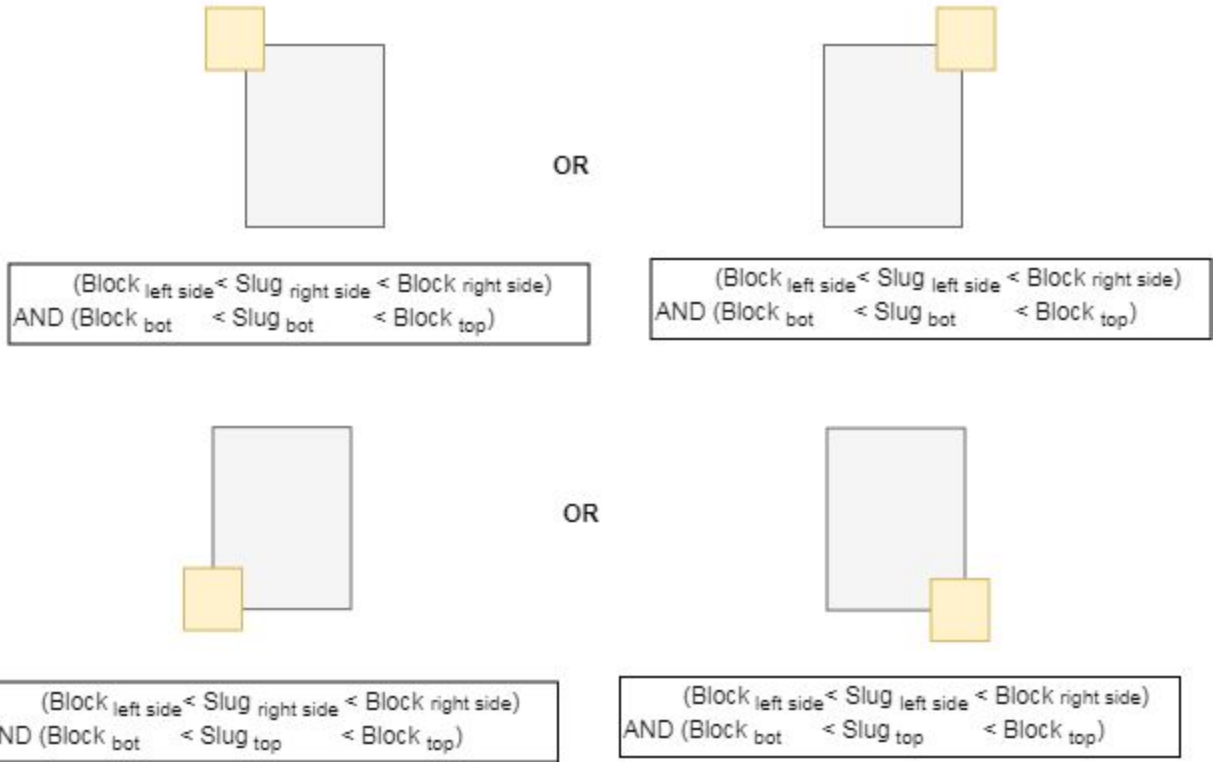
Hit Detection (Output: "onObstacle"): The hit detection was also a single bit signal that was either true or false corresponding to whether the slug was touching an obstacle or not. The slug's coordinates were compared with each obstacles coordinates to determine whether they were oversecting. A diagram below shows all the comparator logic for determining if the slug is overlapping an obstacle.

```
assign withinLeftAndRight = ((slugLeft>obstacleLeft  & slugLeft<obstacleRight) | (slugRight>obstacleLeft & slugRight<obstacleRight));


assign hitObstacle0 =  withinLeftAndRight & ((slugTop>obstacleBot0  & slugTop<obstacleTop0) | (slugBot>obstacleBot0 & slugBot< obstacleTop0));
assign hitObstacle1 =  withinLeftAndRight & ((slugTop>obstacleBot1  & slugTop<obstacleTop1) | (slugBot>obstacleBot1 & slugBot< obstacleTop1));
assign hitObstacle2 =  withinLeftAndRight & ((slugTop>obstacleBot2  & slugTop<obstacleTop2) | (slugBot>obstacleBot2 & slugBot< obstacleTop2));
assign hitObstacle3 =  withinLeftAndRight & ((slugTop>obstacleBot3  & slugTop<obstacleTop3) | (slugBot>obstacleBot3 & slugBot< obstacleTop3));

assign onObstacle = hitObstacle0 | hitObstacle1 | hitObstacle2 | hitObstacle3;
```

4 Ways to Overlap Obstacle



Obstacle Movement: The obstacle column used 1 counter for the entire column to move from the right side of the screen to the left. All of the obstacles had the same left and right wall values, so the counter value controlled all of them. The counter had a hard coded value of 664, which was right off the visible section of the screen. It's Load Control was set to the values provided by the obstacle state machine OR a reset for when the obstacle hit the left side of the screen.

```
countUD16L obstacleMove( .clk(clk), .up(1'b0), .down(frame & run), .loadControl(btnC | obstacleReset), .reset(1'b0),
                         .loadData( {6'b0000 ,10'd664} ), .upperLimit(), .lowerLimit(), .outQ(obstacleRight));

assign obstacleLeft = obstacleRight - 10'd24;
```

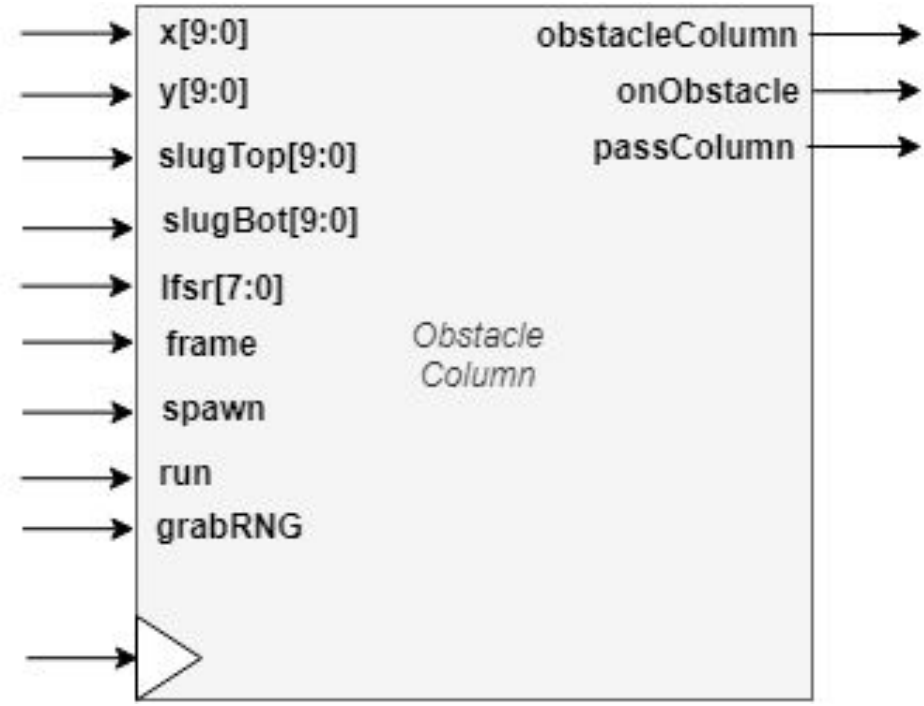
Obstacle Random Offset: An LFSR was designed in a previous lab as a random number generator. A 7 bit register held the value of the offset for the obstacle column. The grabRNG signal went high just before spawning each column and would grab a random value during the initial spawn sequence. The column would then grab a new offset each time it reset when hitting the left side of the screen.

```
Register_4bit rngHold00(.inData(lfsr[3:0]), .clk(clk), .enable(obstacleReset | grabRng), .reset(1'b0), .outQ(rng[3:0]));
Register_4bit rngHold01(.inData(lfsr[7:4]), .clk(clk), .enable(obstacleReset | grabRng), .reset(1'b0), .outQ(rng[6:4]));

assign obstacleBot0 =  obstacleBot + {3'b000,rng};
assign obstacleBot1 =  obstacleBot + {3'b000,rng} + 10'd128;
assign obstacleBot2 =  obstacleBot + {3'b000,rng} + 10'd256;
assign obstacleBot3 =  obstacleBot + {3'b000,rng} + 10'd384;

assign obstacleTop0 =  obstacleTop + {3'b000,rng};
assign obstacleTop1 =  obstacleTop + {3'b000,rng} + 10'd128;
assign obstacleTop2 =  obstacleTop + {3'b000,rng} + 10'd256;
assign obstacleTop3 =  obstacleTop + {3'b000,rng} + 10'd384;
```

Results:



Kyle Jeffrey
Prof Schlag
CE 100
2/21/18
16

Top Level Design:

The top level module connected all of the previously mentioned parts.

Method:

Drawing: All of the drawing was done by telling the RGB signals what to draw, otherwise the entire screen was black. The LCD screen drew pixel by pixel from left to right, so the screen needed to know at each pixel what color to draw in 12 RGB bits. Below is an example of what one of the RGB signals looked like. Slug was defined by ranges within the counters. The aR, stands for “active region” and tells the signal to not draw anything outside of the active region.

```
assign vgaGreen[3:0] = {4{ aR & slug} } ;
```

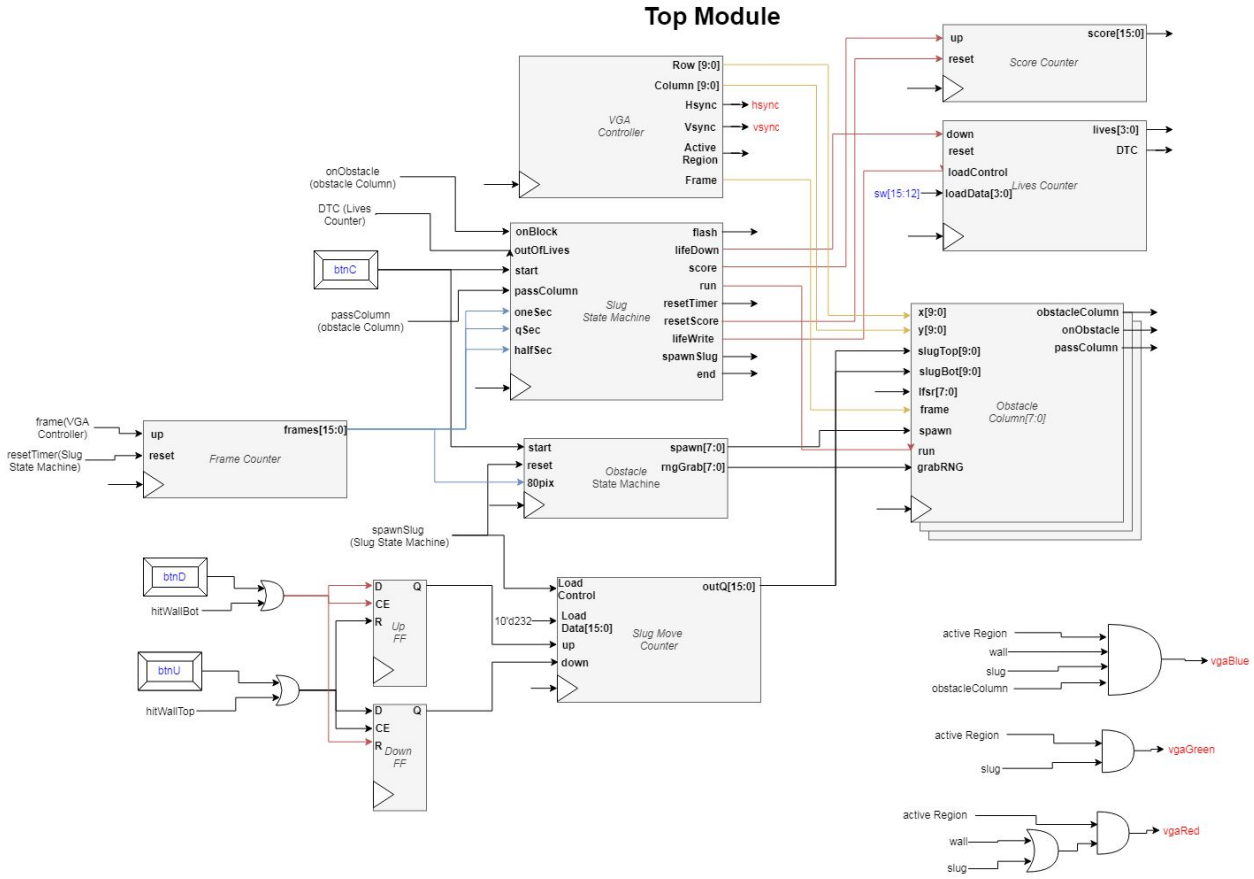
```
assign slugRegular =  (y>slugBot & y<slugTop) & (x>slugLeft & x<slugRight);
assign slugFlash   =  (y>slugBot & y<slugTop) & (x>slugLeft & x<slugRight) & qsec;
mux2_1 slug(.in0(slugRegular), .in1(slugFlash), .sel(flash), .o(slug));
```

To get the slug to flash, a 2 to 1 mux was controlled by the flash signal from the slug state machine. The slug was AND’d with qsec, to mask it every quarter second.

Display Logic: The hex display logic is boilerplate from previous labs and uses the same ring counter, selector, and hex display logic to display the score, and lives.

Results:

See next page.



NOTE: ALL blue signals are inputs and all red outputs. The diagram excludes the display logic for hex displays for readability, as it was the same logic from previous labs. The vgaRGB signals were sign extended to 4 bits after the logic went through.

Conclusion

This lab was a huge leap in difficulty and overall scope. I learned how to build a vga controller and how LCD screens use the signal from a vga port. If I were to redesign any component, it'd be the obstacle columns. I would've found a better way to spawn them, probably by starting them off the correct distance away from each other for the first spawn sequence. I would also design a state machine for the slug movement.

Kyle Jeffrey
Prof Schlag
CE 100
2/21/18
18

Fastest Clock speed the circuit can run at?

The longest delay in the circuit is 8.32 nS. The design could run with a 120 MHz clock, approximately 5 times faster than the current design.