CPRE 281 Final Project: VGA GPU

Jonathan Hess

For my project I decided to create a VGA module that could output to a monitor. I started just trying to get a simple color onto the screen. To do this I looked up what the VGA protocol looks like as well as what would be needed to get the syncing correct.
This was the easiest step because most of the information is readily available.

https://blog.thomaspoulet.fr/assets/content/VGA/logic1.png
https://blog.thomaspoulet.fr/assets/content/VGA/bsferror.png
https://blog.thomaspoulet.fr/bit-banged-vga/
http://lslwww.epfl.ch/pages/teaching/cours_lsl/ca_es/VGA.pdf
https://web.mit.edu/6.111/www/labkit/vga.shtml

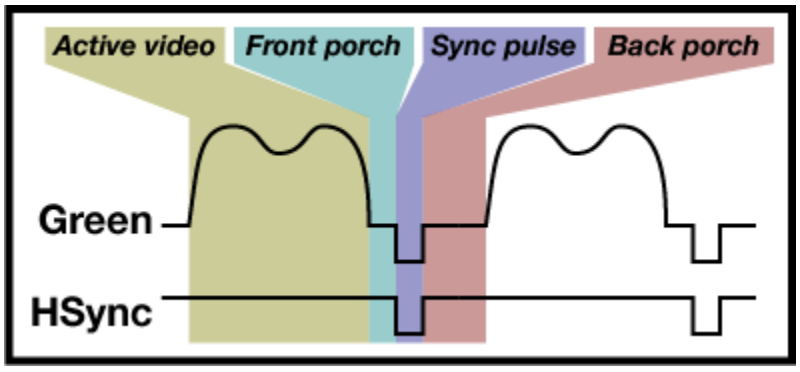
With this knowledge I had to create the pixel clock. Thankfully after some time I learned that the 25.175 mhz pixel clock doesn't have to be completely accurate and simply dividing the 50mhz would suffice. I created a simple verilog divider but I could have used a simple T flip flop instead. I was originally planning on allowing the user to change their resolution (and which they would also change the frequency making this module do more than just divide by 2).

```
module frequency_divider_by2 ( clk ,rst,out_clk );
output reg out_clk = 0;
input clk ;
input rst;
always @(posedge clk)
begin
if (~rst)
     out_clk <= 1'b0;
else
     out_clk <= ~out_clk;
end
endmodule
```

Next was the H_sync and H_blank. Originally, I had them as separate modules but it makes a lot more sense to combine them into one counter.

```
module Horizontal(clk, h_sync, h_blank);


   input clk;
   output reg h_blank;
   output reg h_sync;
   integer counter = 0;

 always @(posedge clk) begin
    counter = counter + 1;
    if(counter <= (16))begin
      h_blank = 0;
      h_sync = 1;
    end

    else if(counter <= (16+96))begin
      h_blank = 0;
      h_sync = 0;
    end
    else if(counter <= (16+96+48))begin
      h_blank = 0;
      h_sync = 1;
    end

    else if(counter <= (16+96+48+640))begin
      h_blank = 1;
      h_sync = 1;
    end



    if(counter >799) begin //reset counter every line
      counter =0;
      end

  end
endmodule
```

It works as a simple counter that resets after 800 clock cycles (or one line on the monitor).



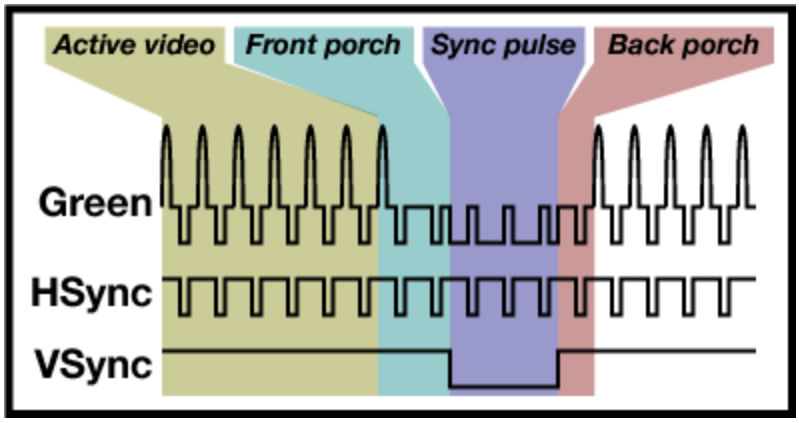
During these it pulls the h_sync high except in the 96 cycles after the front porch and pulls the h_blank low when not in active video.

This module could be implemented with a simple counter with 9 bits (d flip flops) instead of using verilog.

The Vsync is similar to how it computes the vsync and v_blank.

```
module vSync(h_sync, v_sync);


   input h_sync;
   output reg v_sync;
   integer counter = 0;

 always @(posedge h_sync) begin
    counter = counter + 1;
    if(counter >=(480+11) && counter <=(480+11+2)) begin //deactivated when gre
      v_sync = 0;
      end else begin
      v_sync =1;
      end


      if(counter >523) begin //reset counter every line
      counter =0;
      end
  end
endmodule
```

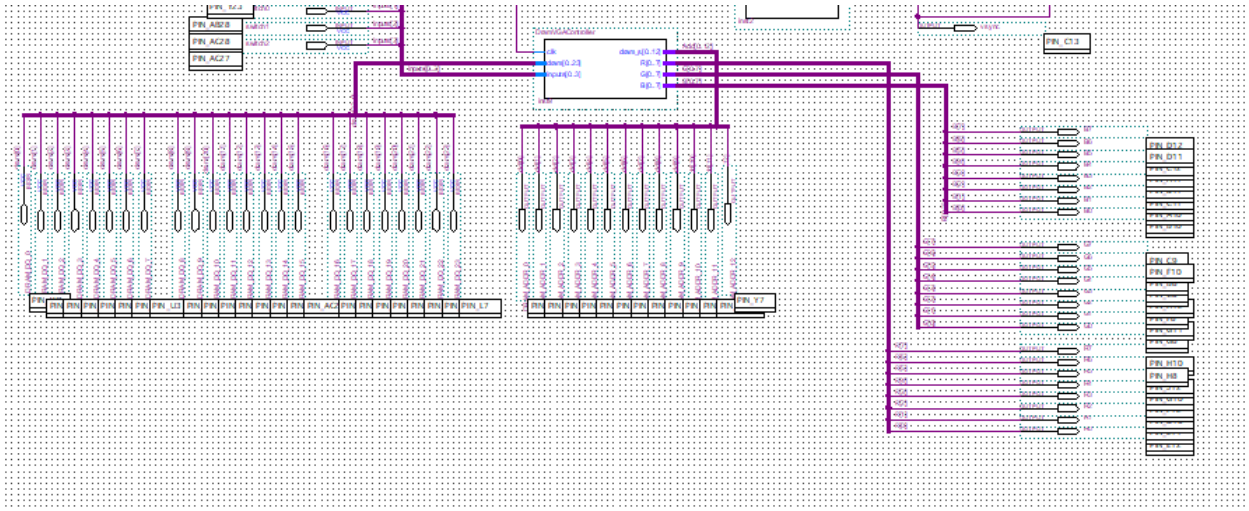
```
module vBlankVertical(clk, v_blank);


   input clk;
   output reg v_blank;
   integer counter = 0;

 always @(posedge clk) begin
    counter = counter + 1;
    if(counter >=(480) && counter <=(480+11 +2 + 31) ) begin //deactivated
      v_blank = 0;
      end else begin
      v_blank =1;
      end


      if(counter >523) begin //reset counter every line
      counter =0;
      end
  end
endmodule
```



With all of this I could set the pins of RGB and have a colored screen with 8 options!

Video of basic colors:
https://drive.google.com/file/d/1dKky_hVdC9blb6mkqQ52f_EU8sofT8pk/view?usp=sharing

After this I tried to get the loading off the SDRAM (Synchronized Dynamic Ram). I couldn't get to read anything (especially because I hadn't written anything to it). I was hoping that it would show some random colors because it would be random but it was either zeroed out or didn't work. The reason I wanted to use the SDRAM was because it would allow me to do things such as layering and making text simpler because I could update the screen rather than rendering a new one each frame.



This never worked so I could not implement hire functions like text and imaging as easily, (I will work on this after the class in the GPU module that is more instruction based)

CHARACTER MODE

Before pitching the project I worked on one of my end goals of getting text on the screen. I chose the iconic Microsoft Latin font because it was iconic and was simple with how it divided the characters. Things like kerning would be considered if I had excess time.

This was the only way I could find the font which meant I would have to create my own hex file for it.

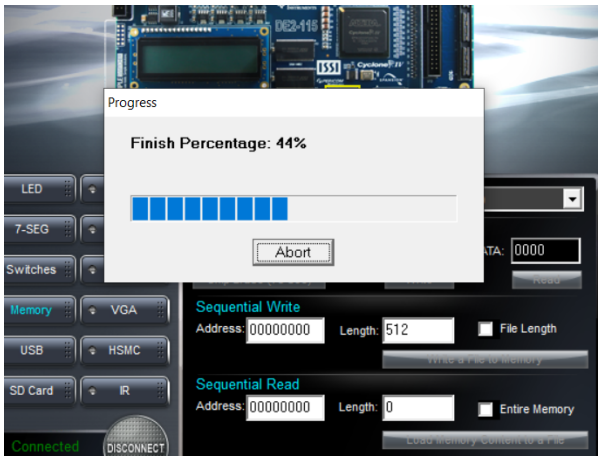
To create a hex file I could use in my project I created this python script to turn the font image into a hex file. It is shown below.https://hexed.it/

```
import numpy as np
from numpy import asarray

import cv2
import PIL
from PIL import Image


image = Image.open('Codepage-437.png')
data = asarray(image)
im = np.array(Image.open('Codepage-437.png').convert('L')) #you can pass multiple arguments in single line
print(type(im))
image.show();
hexString = "";
for yNum in range(8): #8
    for xNum in range(32): #32
        print("new char");
        x = 8+xNum*9;
        y = 8+16*yNum;
        height = y+16;
        width = x +8;
        img = image.crop((x,y,width,height))
        imA = np.array(img.convert('L'));
        Image.fromarray(imA).show();
        rows = imA.shape[0]
        print("rows ")
        print(img.size[1]);
        cols = imA.shape[1]
        print("cols ")
        print(img.size[0]);

        hexOut = []
        pixels = img.load();
        for i in range(img.size[1]): # for every pixel:
            hexInt = 0
            for j in range(img.size[0]):
                if pixels[j,i] >= 100:
                    hexInt += 1<<j;
            #print(bin(hexInt));
            if(len(hex(hexInt)[2:]) == 1):  hexString +="0";
            hexString +=hex(hexInt)[2:]
            hexOut.append(hex(hexInt));
print(hexOut);
print(hexString);

        #img.show();

file2write=open("hex",'w')
file2write.write(hexString)
file2write.close()
```

Each character is 8 by 16 pixels and there is an 8 pixel buffer. It will be stored in the SRAM because it doesn't change over the course of the program. A small portion looks like this.

```
00000000000000000000000000000007e81a58181bd9
981817e0000000000007effdbffffc3e7ffff7e0000000000
000000367f7f7f7f3e1c0800000000000000081c3e7f3e1
c0800000000000000183c3ce7e7e718183c000000000000
00183c7efff7e18183c0000000000000000183c3c180
00000000000fffffffffffe7c3c3e7fffffffffffff000000
00003c664242663c0000000000ffffffffffc399bdbd99c3f
fffffffff00007870584c1e333333331e000000000003c66
6666663c187e1818000000000000fcccfc0c0c0c0c0e0f070
00000000000fec6fec6c6c6c6e6e76703000000000001818
db3ce73cdb1818000000000000103070f1f7f1f0f070301000
```

The hardest part of this project is finding information on how to access both the SRAM and SDRAM. One guide recommended using the control panel that comes with the Altera board (or can be found online here: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502&PartNo=4)
This sadly wasn't on the lab computers so I had to manually download it. Another problem is that the control panel was only supported by the 32 bit drivers. The lab computers also did not have these drivers and I couldn't install them without admin permissions. Instead I got Quartus on my own personal computer and installed the drivers there. Then I could load the font files on my personal computer then program the board with the lab computer. The control panel looks like this:

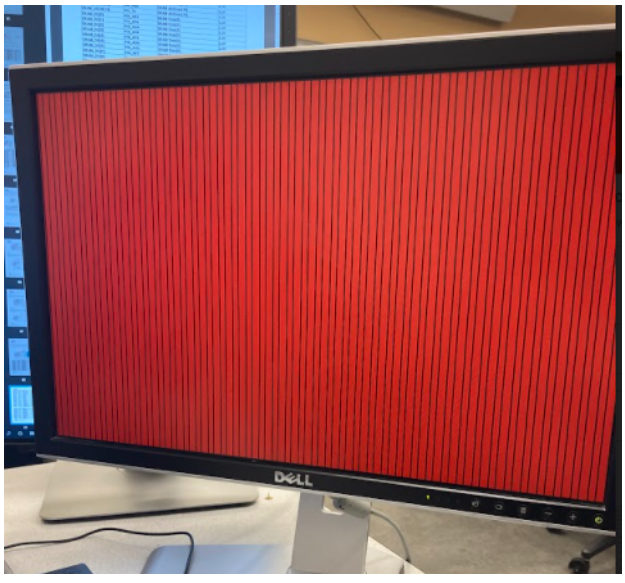


It has the ability to load both the SRAM and SDRAM without having to implement my own personal loader.

After checking that the rom worked by testing the data of one byte of the characters, I got

```
module SRAMCharController(clk, row, char, SRAM_ADDR,SRAM_DQ, binChar, inputs,h_blank,v_blank);
    input clk;
    output reg [0:19] SRAM_ADDR = 0 ;
    input [0:11]inputs;
    input [0:15] SRAM_DQ;
    input [0:7] char;
    input [0:9] row;
    input h_blank;
    input v_blank;
    reg [0:3] charCounterHor;
    reg [3:0] charCounterVer;
    integer vCount;
    integer hCount;
    output reg binChar;
    integer counter = 0;
always @(row) begin
        charCounterVer = row%16;
    end


    always @(posedge clk) begin
        //SRAM_ADDR = {1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,1'b0,inputs};
        //binChar = SRAM_DQ[counter];
        //counter = counter + 1;

        //if(counter>15) begin counter = 0; end

        charCounterHor = charCounterHor +1;


        SRAM_ADDR = {1'b0,1'b0,1'b0,1'b0,1'b0,inputs,charCounterVer[3],charCounterVer[2],charCounterVer[1]};

        if(charCounterVer[0])begin
        if(charCounterHor<8) begin
            binChar = SRAM_DQ[charCounterHor+8];
            end
            else begin
            binChar = 0;
            end
            //binChar = 0;
        end
        else begin
        if(charCounterHor<8) begin
        binChar = SRAM_DQ[charCounterHor];
        end
            else begin
            binChar = 0;
            end
        end
    end


endmodule
```

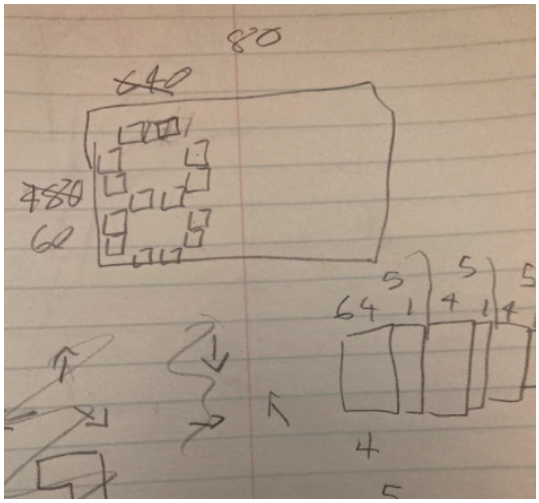
A fun accident that happened while trying to display is shown below.
https://drive.google.com/file/d/1K8CLmRVzxOAjKKJAiyxzLQ4kRCJjGuhm/view?usp=sharing



Originally I used a simpler program to look at singular words of data in order to see if the SRAM was loading correctly. This is an image from that test. The horizontal lines are the only thing changed as they check the bytes binary values (0 is black pixel, 1 is red)

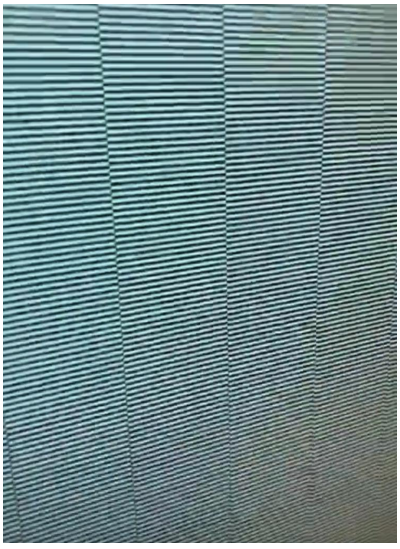
The final result looked like this, it fills the screen up with whatever character is selected with the switches.
https://drive.google.com/file/d/1p8HliQSmONKqsRppa8V8IzgmA9m7iLuC/view?usp=sharing

7 SEGMENT MODE



Dividing pixels idea inspired by this website: https://blog.thomaspoulet.fr/bit-banged-vga/

The 7 segment display mode is probably the most useful mode of the project because it allows you to use the monitor as an extension to the existing 7 segment displays.
I first started out with my plan.
I had a resolution of 480*640 which I would want to divide the resolution by 32 to make it simpler. This gives a new resolution of 15 by 20 new pixels. With 2 (32x32) new pixels per segment we get a length of 4 new pixels (4*32x32 pixels) per module. With some buffer of 1 new pixel width per segment you get a length of 5 new pixels. Because the resolution is 20 this gives you 4 seven segments.



We need 4 bits to store the height (15) (possibly not because we are only using the 7 of the 15 height) and 5 to store the width (20). With 5 t flip flops on the hsync, clock and v_sync we can divide their frequency by 32.

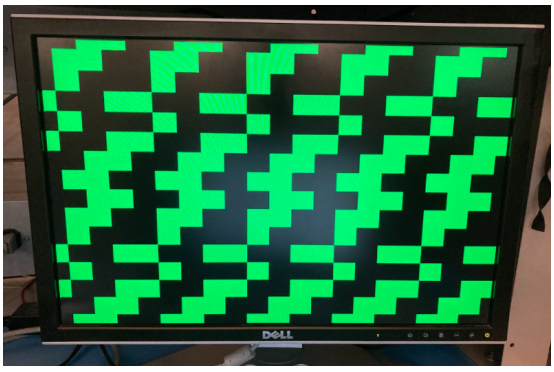
First thing was to create a 32 bit clock divider. I used the one shown in lab 11 and removed half of them to keep 5 stages of t flip flops. (synchronous). Then I displayed the result with a t flip flop. The result is shown on the left.
Video of the screen above: horizontal divided by 32

Then I made it reset every horizontal blank clock to get this (shown on the left).

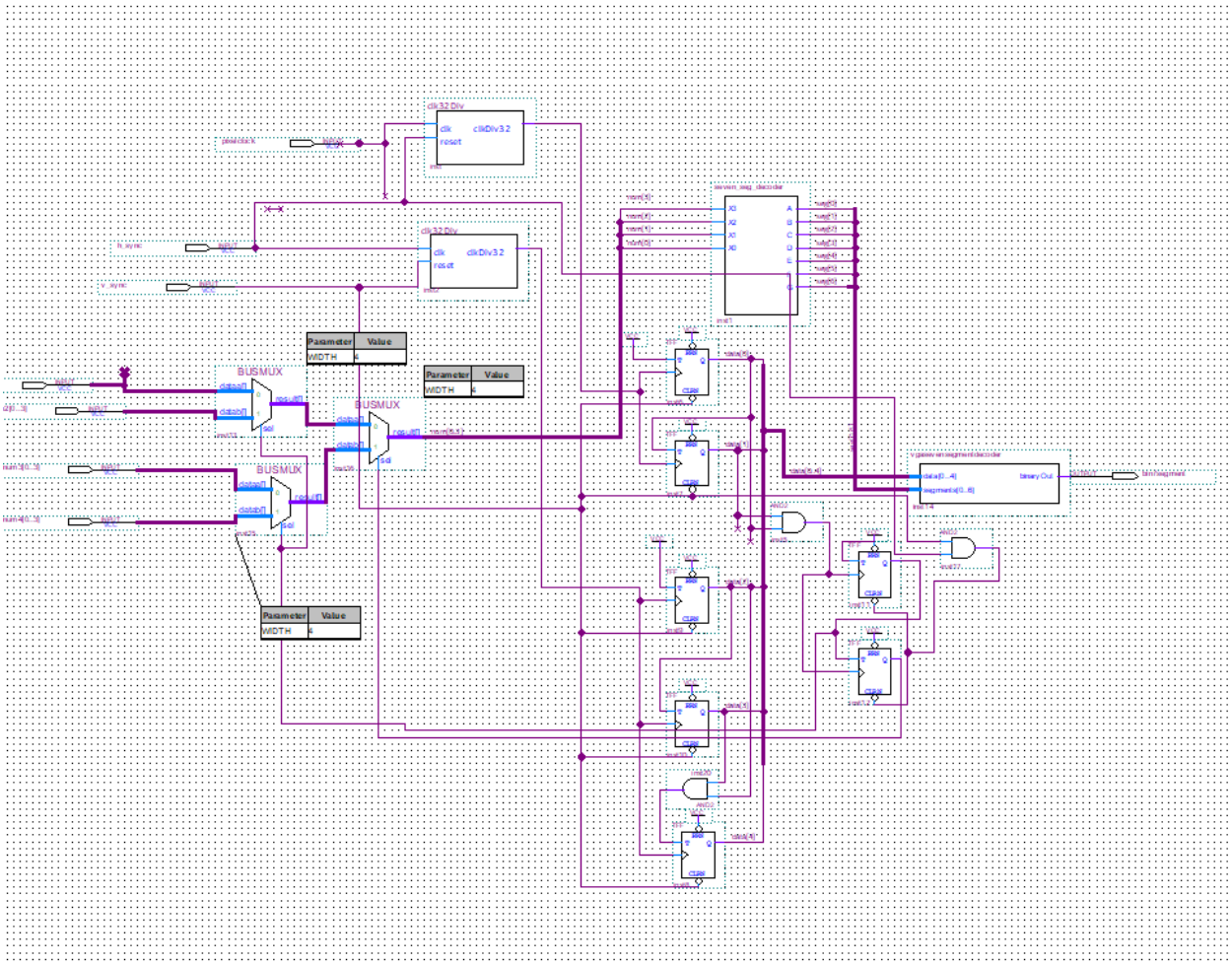
I then split the horizontal in a similar way and implemented counters. (one that counts 0-3 for horizontal and the other 0-7 for vertical)
Finally I was up to creating the pixels segments

```
module vgasevensegmentdecoder(data, segments, binaryOut);
    input [4:0] data;
    input [6:0] segments;

    output reg binaryOut;

    always @(data or segments) begin
        case(data)
            5'b00101: binaryOut = ~segments[0]; //A
            5'b00110: binaryOut= ~segments[0];

            5'b01011: binaryOut= ~segments[1]; //B
            5'b01111: binaryOut= ~segments[1];

            5'b10111: binaryOut= ~segments[2]; //C
            5'b11011: binaryOut= ~segments[2];


            5'b11101: binaryOut= ~segments[3]; //D
            5'b11110: binaryOut= ~segments[3];


            5'b10100: binaryOut= ~segments[4]; //E
            5'b11000: binaryOut= ~segments[4];


            5'b01000: binaryOut= ~segments[5]; //F
            5'b01100: binaryOut= ~segments[5];

            5'b10001: binaryOut= ~segments[6]; //G
            5'b10010: binaryOut= ~segments[6];
            default: binaryOut = 0;
        endcase

    end

endmodule
```

Seven segment mode before splitting into 3 different segments



One problem that took me a long time to solve was that I had forgotten that the seven_segment_decoder was active on 0 rather than 1. This meant my 8 was a blank screen. After solving that it worked almost just as designed! The only problem now is that the screen was offset 16 pixels. This is the whole 7 segment module with the binary out.

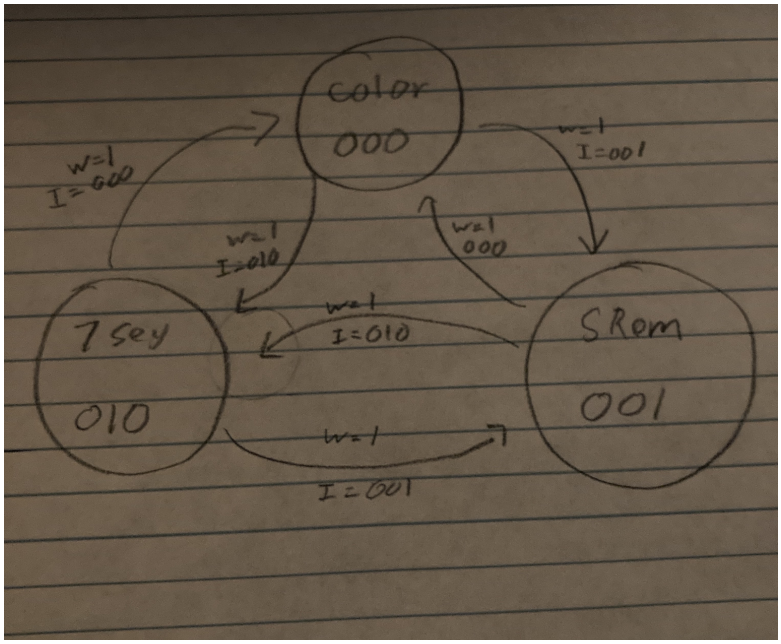


This binary signal goes into the colorizer where it uses the color stored from the color mode.

```
module colorizer(clk, bin, color_flag, iR,iG,iB,R,G,B);
input clk;
input bin;
input color_flag;

input [0:3] iR;
input [0:3] iG;
input [0:3] iB;

reg [0:7] sR;
reg [0:7] sG;
reg [0:7] sB;

output reg [0:7] R;
output reg [0:7] G;
output reg [0:7] B;

always @(posedge clk) begin
if(color_flag) begin



 sR = {iR[0],iR[0],iR[1],iR[1],iR[2],iR[2],iR[3],iR[3]};
 sG = {iG[0],iG[0],iG[1],iG[1],iG[2],iG[2],iG[3],iG[3]};
 sB = {iB[0],iB[0],iB[1],iB[1],iB[2],iB[2],iB[3],iB[3]};

R = sR;
G = sG;
B = sB;
end
else begin

if(bin)begin
R = sR;
G = sG;
B = sB;
end
else begin
R = 0;
G = 0;
B = 0;
end
end


end


endmodule
```

The colorizor only changes when in the color setting.

The finite state machine was very simple because I couldn't get the SDRAM to work for more complex instructions. It used the rocker switches to change modes, you press the button when you want to change. Any input other than the 2 non 000 will be the same as a 000 input.



$w\ I_2\ I_1\ I_0$	$S_2\ S_1\ S_0$
0 0 0 0	$S_2\ S_1\ S_0$
1 0 0 0	0 0 0 – color
1 0 0 1	0 0 1 – SRam
1 0 1 0	0 1 0 – 7segment
1 0 1 1	0 0 0
1 1 0 0	0 0 0
1 1 0 1	0 0 0
1 1 1 0	0 0 0
1 1 1 1	0 0 0

S_2

I_1I_0 \ wI_2	00	01	11	10
00	S_2	S_2	0	0
01	S_2	S_2	0	0
11	S_2	S_2	0	0
10	S_2	S_2	0	0

$S_2 = 0$

S_1

I_1I_0 \ wI_2	00	01	11	10
00	S_1	S_1	0	0
01	S_1	S_1	0	0
11	S_1	S_1	0	0
10	S_1	S_1	1	1

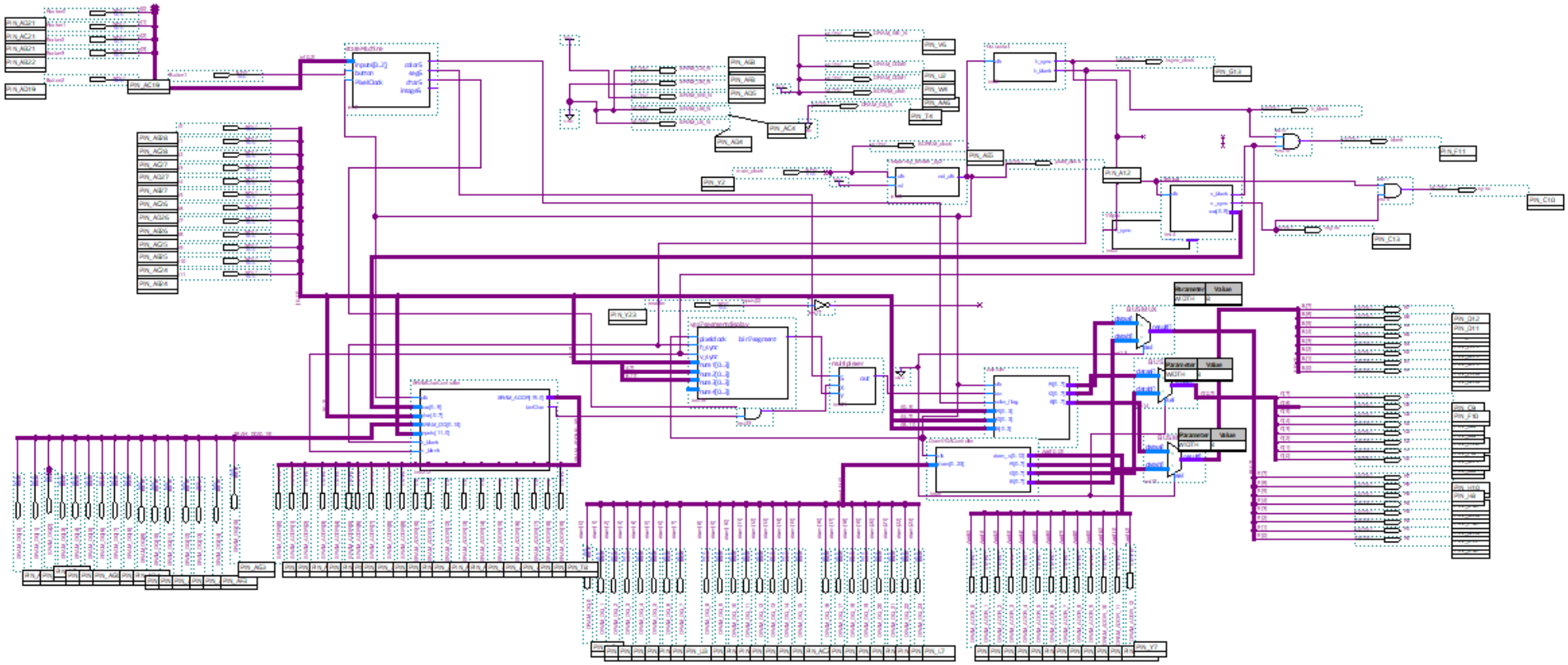
$S_1 = \bar{w} S_1 + w \bar{I}_2 I_1 \bar{I}_0$

$S_0 I_1 I_0$ \ wI_2

	00	01	11	10
00	S_0	S_0	0	0
01	S_0	S_0	0	1
11	S_0	S_0	0	0
10	S_0	S_0	0	0

$S_0 = \bar{w} S_0 + w \bar{I}_2 I_1 \bar{I}_0$

All together it looks like this.



The 2 AND symbols are used for the blank and sync so that it pulls the blank or sync when either blank or sync is 0. The muxes on the right hand side are used for switching between the colorizer and the SDRAM. I couldn't get the SDRAM to work properly so I hard coded it to the colorizer.

USER MANUAL:
When you boot up the VGA Trial onto the board you must connect the board to an external monitor using a VGA cable.
The board always starts in state 000. By changing the rocker switched on the right external board you can change the mode selected. Then by pressing the push button bellow the rocker switches you will change modes.
If in the mode 000 you will be in the color mode it will use the first 4 switches (0-3) for the red value, the next 4 for green (4-7) and the last 4 for blue (8-11). This changes the color of the entire screen. If you leave this mode the rest of the VGA Trial will use whatever color you selected for other modes.

If in the mode 010 you will be in the 7 segment display mode where the value of the 3 seven segments is equal to the 4 switches corresponding to them. (0-3 for the 1st, 4-7 for the 2nd, 8-11 for the 3rd)

If in the mode 100 you will be in the character mode where the value of the first 8 switches (0-7) will select the ascii value (NOTE: THIS MODE ONLY WORKS WHEN FONT FILE IS INSTALLED INTO SRAM)

Useful links:
https://blog.thomaspoulet.fr/assets/content/VGA/logic1.png
https://blog.thomaspoulet.fr/assets/content/VGA/bsferror.png
https://blog.thomaspoulet.fr/bit-banged-vga/
https://blog.thomaspoulet.fr/bit-banged-vga/
http://lslwww.epfl.ch/pages/teaching/cours_lsl/ca_es/VGA.pdf
https://web.mit.edu/6.111/www/labkit/vga.shtml