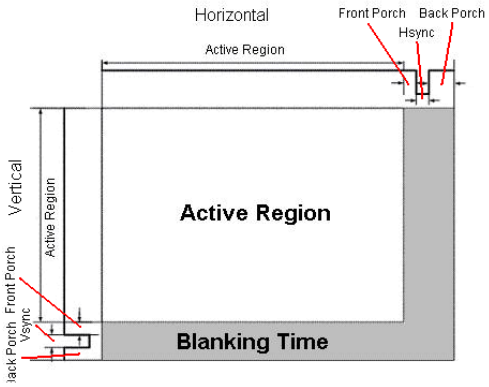
CSE3017 PROJECT – VGA CONTROLLER

1. The Requirements

The controller needs to take 8-bit Character Codes which is a custom set with simple ASCII compatibility from a RAM which would be written by the CPU or any other module and output these as a VGA Signal for a VGA Inputting display to use and visualize. There will also be 3-bit Character Color and 3-bit Background Color data in the same RAM word. The RAM is 16-bits in word size and 12-bits in address width, a total of 4096 Words. The part where VGA controller will use will be reserved as the VRAM and the rest will be open for any other modules to use. The VGA controller will not write any data to RAM but will only read from its designated VRAM.

2. How VGA Works

VGA is an analog video transmission standard which is widely used and roots its design from how old CRT Displays work. Each frame of the video will be constructed one line at a time and would start from top-left corner. In CRT Displays this is done with an electron beam guided using some electromagnets to hit the display that draw the pixels. But today we mostly use LCD Displays, and the LCD Display's have their own way to decode these VGA signals. The Voltage Levels, defined by the specification, are listed below table.



Signal	Logic Voltage	On-Board Pin
VSYNC	5V	PIN_103
HSYNC	5V	PIN_101
R	0.7V	PIN_106
G	0.7V	PIN_105
B	0.7V	PIN_104

The VGA Display uses HSYNC and VSYNC Signals to synchronize with the VGA controller. Signaling the end of the line and the end of the frame respectively. These signals change a lot depending on resolution or the refresh time of the outputted video. These signals are modulated with time, changing their values in certain time frames, these limitations for each standard can be calculated/found on internet. HSYNC and VSYNC signals are logic 1 always but when it's time to "sync". On the other hand, Color Signals are default 0 and can be pulled to logic 1 to create certain colors at the same time if in it's in the "Active Region". The colors should be all 0 in the "Blanking time". Sync signals have a front-porch and back-porch with logic 1, probably to protect old CRT Display circuitry.

3. Choosing the specific values for the requirements.

We need to choose what VGA standard to use and how to slice the total pixels so draw each character. I used this website to explore each specification: http://tinyvga.com/vga-timing

Our FPGA has a clock of 50MHz, therefore first thing I would look for would be a 50Mhz pixel clock standard. And the only standard would be the 800x600@72 Hz. Even though the resolution and the pixel clock were OK. The refresh rate of 72Hz would be too much for my monitor or any common 60Hz display. That's why the next best choice would be the 640x480@60 Hz Industry standard with a pixel clock of 25.175MHz. I will be dividing my 50Mhz clock to 25Mhz and use that for the pixel clock, even if not perfect it fits just enough. And that's how I chose the **640x480@60 Hz Industry standard** as the target. All the timing data can be found on the website.

Now, I decided to divide 640x480 resolution to 8x16 rectangular slices and draw the each character to these rectangular sites, I've found a fitting font and converted it into a 16384 bit length 1bit wide ROM. The controller will be reading from this font ROM each 2 clocks or each pixel clock to check if we are in the character or the background. The ROM data file can be found in the project folder.

Now that we decided how we will slice the display we can determine how much of the RAM we will be designated as the VRAM. 640/8 = 80 characters per line and 480/16 = 30 lines for the total frame, that would mean a 80*30 = 2400 characters. Which corresponds to a 2400 word of RAM and 2400*16 = 38400 bits in total space.

4. Designing the Synchronization module (Verilog)

```
module HVSync(
// 25 Mhz clock
input clk25MHz,

output reg hsync,
output reg vsync,
output reg inDisplayArea,
// Module Interfaces
output reg [9:0] counterX,
output reg [9:0] counterY
);

wire counterXmaxed = (counterX == 800); // 16 + 48 + 96 + 640
wire counterYmaxed = (counterY == 525); // 10 + 2 + 33 + 480
```

```
always @(posedge clk25MHz)
begin
if (counterXmaxed)
begin
counterX <= 0;
end
else
begin
counterX <= counterX + 1;
end
end

always @(posedge clk25MHz)
begin
if (counterXmaxed)
begin
if(counterYmaxed)
begin
counterY <= 0;
end
else
begin
counterY <= counterY + 1;
end
end
end

always @(posedge clk25MHz)
begin
hsync <= ~(counterX > (640 + 16) && (counterX < (640 + 16 + 96))); // active for
96 clocks
vsync <= ~(counterY > (480 + 10) && (counterY < (480 + 10 + 2))); // active for 2
clocks
end

always @(posedge clk25MHz)
begin
inDisplayArea <= (counterX < 640) && (counterY < 480);
end
endmodule
```

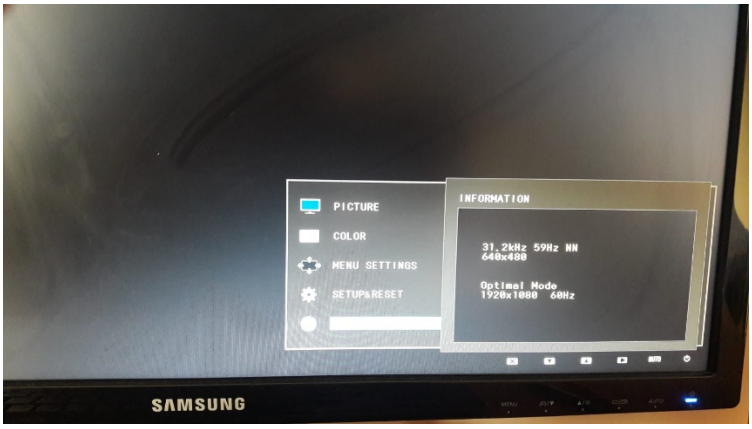
5. Designing the VGA Controller module (Verilog)

```
module VGAController(
// 50 Mhz clock
input clk,

output hsync,
output vsync,
output [2:0] rgbOut,

// Clock Reset
input rst,

output [13:0] ROMAddr,
input ROMOut,
output [11:0] RAMAddr,
input [15:0] RAMOut
);

wire inDisplayArea;
wire [9:0] PosX;
wire [9:0] PosY;

// 25Mhz Clock generator
reg clk25MHz;
always @(posedge clk)
begin
if (~rst)
clk25MHz <= 1'b0;
else
clk25MHz <= ~clk25MHz;
end

wire hsyncOrig, vsyncOrig, inDisplayAreaOrig;

HVSync(clk25MHz, hsyncOrig, vsyncOrig, inDisplayAreaOrig, PosX, PosY);

// Drawing Logic goes here
wire [7:0] asciiData;
wire [2:0] colorF;
wire [2:0] colorB;

assign asciiData = RAMOut[15:8];
assign colorF = RAMOut[7:5];
assign colorB = RAMOut[4:2]; // 2 bit unused for now...

// (PosX / 8) + (PosY / 16) * 80 Text Indexing
assign RAMAddr = {5'b00000, PosX[9:3]} + ({6'b000000, PosY[9:4]} * 80);
```

```
reg [2:0] CharX;
reg [3:0] CharY;
always @ (posedge clk) // Adds 1 clock delay to sync, compensating the RAM delay.
begin
CharX <= PosX[2:0];
CharY <= PosY[3:0];
end

assign ROMAddr = {asciiData[6:0], 7'b0000000 } + CharX[2:0] + {CharY[3:0],
3'b000}; // Text Pixel Indexing

assign rgbOut[0] = ROMOut ? (colorF[0] & inDisplayArea) : (colorB[0] &
inDisplayArea);
assign rgbOut[1] = ROMOut ? (colorF[1] & inDisplayArea) : (colorB[1] &
inDisplayArea);
assign rgbOut[2] = ROMOut ? (colorF[2] & inDisplayArea) : (colorB[2] &
inDisplayArea);


reg hsyncDelayed1, hsyncDelayed2, hsyncDelayed3;
reg vsyncDelayed1, vsyncDelayed2, vsyncDelayed3;
reg inDisplayAreaDelayed1, inDisplayAreaDelayed2;


always@(posedge clk) begin
hsyncDelayed1 <= hsyncOrig;
hsyncDelayed2 <= hsyncDelayed1;
hsyncDelayed3 <= hsyncDelayed2;

vsyncDelayed1 <= vsyncOrig;
vsyncDelayed2 <= vsyncDelayed1;
vsyncDelayed3 <= vsyncDelayed2;

inDisplayAreaDelayed1 <= inDisplayAreaOrig;
inDisplayAreaDelayed2 <= inDisplayAreaDelayed1;
end

assign hsync = hsyncDelayed3;
assign vsync = vsyncDelayed3;
assign inDisplayArea = inDisplayAreaDelayed2;

endmodule
```

6. Practical Testing

Testing of the HVSync module is done by running it with all black (0) pixels from the controller. My Display detects the signal and even gives data about the signal.



As you can see from the image, the signal is detected to be a 640x480@59Hz. Even though we meant to refresh at 60Hz the monitor detected it to be a 59Hz signal. The reason would be that since we used 25Mhz pixel clock instead of the 25.175MHz. This difference causes errors in the sync signals which then results in such a way.

Now I initialized the RAM with following values.

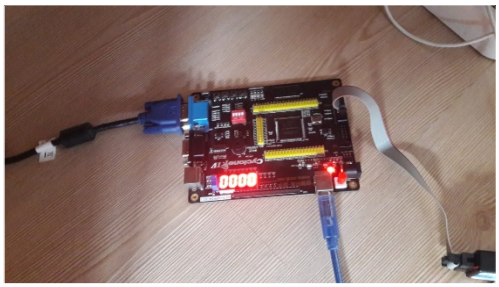
```
1   0100100011101000
2   0110010111101000
3   0110110011101000
4   0110110011101000
5   0110111111101000
6   0010000011101000
7   0111011111101000
8   0110111111101000
9   0111001011101000
10  0110110011101000
11  0110010011101000
```

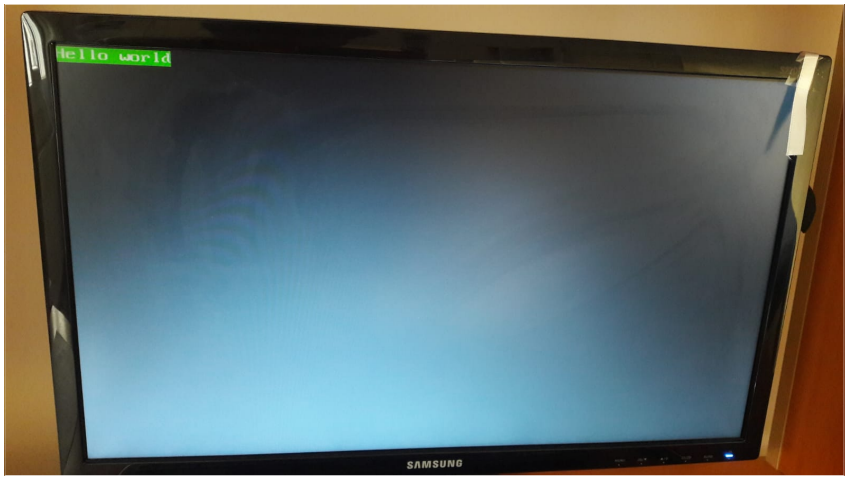
Which corresponds to “Hello World”

with foreground color 111 -> white (red, green, blue)

and background color 010 -> green

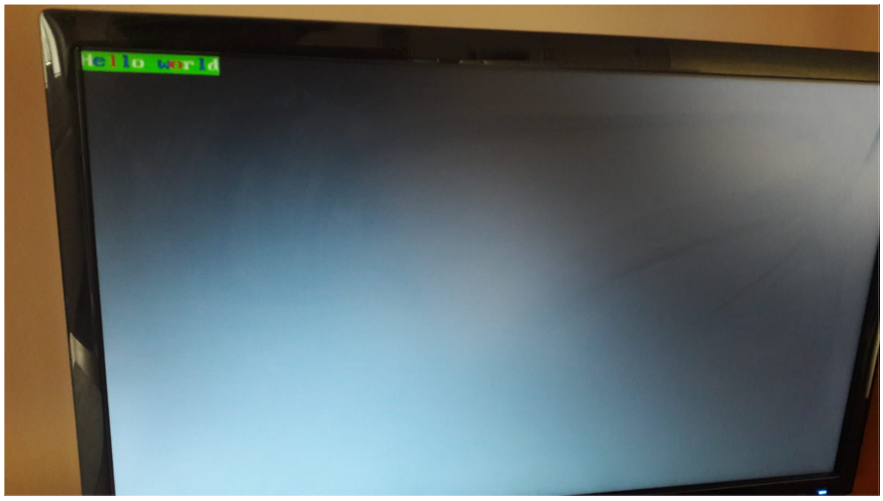
And here is the result:





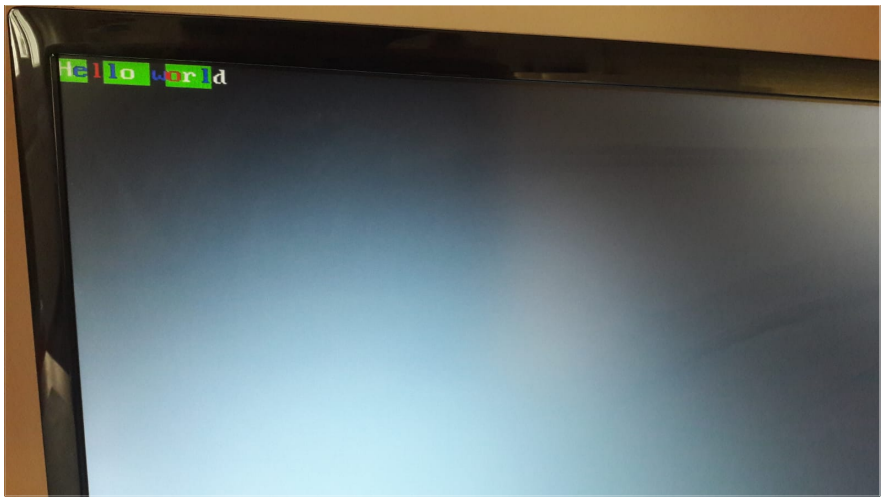
We can change some of the characters color to 001 for blue and some to 100 for red:

```
1   0100100011101000
2   0110010100101000
3   0110110010001000
4   0110110000101000
5   0110111111101000
6   0010000011101000
7   0111011100101000
8   0110111110001000
9   0111001011101000
10  0110110000101000
11  0110010011101000
```



As well as some of the background colors to 000 for black:

```
1    0100100011101000
2    0110010100101000
3    0110110010000000
4    0110110000101000
5    0110111111101000
6    0010000011101000
7    0111011100100000
8    0110111110001000
9    0111001011101000
10   0110110000101000
11   0110010011100000
```



So, It’s fully functional and fully customizable by the data in the RAM.

Netlist Views are also included in the project file If interested in.

Also the link for the demo videos:

(VGATester and Direct Read from RAM)

https://drive.google.com/file/d/1HMef6Q59-3zZ4NQODptuqgn0An9vSW9P/view?usp=sharing

(With CPU writing to RAM):

https://drive.google.com/file/d/1Rim_TL8SQ2SJTGS2j00ESfhPuEdO5mO8/view?usp=sharing