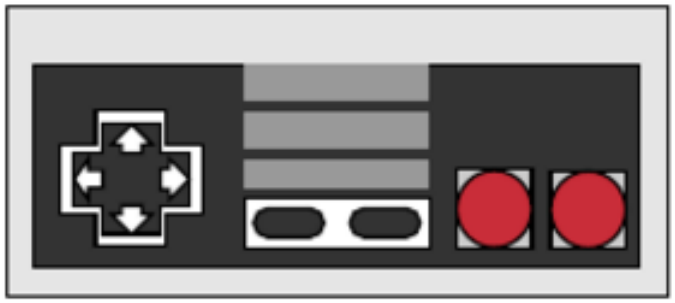
ELEX 7660: Term Project

Nintendo Entertainment System On Chip



Abstract

The Nintendo Entertainment System On chip was an attempt to build a working nes emulator using SystemVerilog on a DE0-NANO terasic development board. The project proved to be a colossal undertaking with a large number of missteps along the way. The end result was a working Picture processing unit capable of rendering graphics from genuine NES memory, and capable of rendering any screen that could be generated from a simple NROM-style nes game such as donkey kong. Additionally we also had a VGA video sync generator that was capable of generating digital video at a resolution of 640x480. Although the entire NES system was not able to be integrated in time, the end result was still quite impressive and leaves roads open for further development.

Prepared by:	Prep Date:
Peter Li	April 18, 2017
Cameron Morgan	April 18, 2017

Contents

List of Figures

1 Introduction

1.1 Motivations

When presented with the opportunity to build a project of some sort in an FPGA, we wanted to take on a project that would offer a lot of insight into how complete systems are integrated and built up as a single chip. So that left us with a few options; to build an entire system from scratch as well as its peripherals and integrate it together, or to take a well understood system that has been well understood and implement our own interpretation of that. To this end, we came to look at the old nintendo entertainment system and sought to rebuild/remix the system using modern technologies. Oftentimes, people complete this task by the creation of many software based emulators. However nowadays with the skills learned in ELEX 7660 we are able to actually build a real hardware implementation of the NES, this would offer a ton of insight into how larger systems are built up out of simple logical constructs.

Another reason for our creation of the NES system is the desire to work on something that is genuinely real. While the NES is not exactly cutting edge, it was once upon a time, quite cutting edge, and though technology has advanced, the same techniques and architectures of systems hasn't changed too drastically. So this project would give us an incredible chance to see how professional logical systems move around.

1.2 Inspirations and other projects

Many of the NES systems have been already reverse engineered and similar projects have been done before in the past. The vast majority of these implementations are done via software in the form of emulators. Ironically, emulators are actually much more difficult to do in software than they are to do in hardware. The reason is because true emulation requires you to simulate multiple hardware components simultaneously which is quite difficult to do efficiently on even modern CPUs. However the use of an FPGA allows us to directly implement each of the hardware components directly so this difficulty is mitigated. Some similar projects include:

- **System On a Chip** - By Steven Hall as part of his 6.111 Final project (Project No. 16 on this page `http://web.mit.edu/6.111/www/f2005/projects/index.html`)
- **Luddes FPGA NES** - An NES implementation in FPGA, done by somebody who is quite skilled, Details are kind of sparse but shows that the project is theoretically possible within our timeline (`http://fpganes.blogspot.ca/2013_01_01_archive.html`)
- **Nesdev.com Programming Guide** - by Brad Taylor, details the programming of an NES emulator and various techniques for doing so. Its heavily oriented on software development but same principles apply to an FPGA implementation. (`http://nesdev.com/NES%20emulator%20development%20guide.txt`)

1.3 What is the NES

The Nintendo Entertainment System (commonly known as the NES) is a gaming console released outside of japan based on the Japanese Famicom system. [1]. The Nintendo NES was released. Sometime In 1987 due to the strict quality assurance performed on the software of the system, the console became one of the most popular and bestselling toys in the world.

2 A simple look at the nes system and components

The Nintendo NES is a huge system and the creation of an entire system capable of emulating the entire system would've been a colossal undertaking that would be fare more work than is reasonable for a half-term project for a group of two.

Initially we wanted implement an entire SOC to encapsulate the entire system and provide emulation for NES ROMs. Partway through the design process we found that such a task was simply not feasible. So our game plan was reduced in scope. Our plans were now to create a VGA output interface, and provide it with graphical data generated from our implementation of an NES Picture processing unit.

While our initial investigations and forays into the titanic amount of information available for the NES had only left us confused and disoriented, we were soon able to distill the fundamental building blocks of the NES into a few simple and easy to understand components.

The key components of the NES are:

- **Cartridge** - the NES rom where all instructions and graphical data is sourced from
- **a203 CPU** - A MOS 6502 based cpu that executes instructions and provides control over the system
- **2c02 PPU** - the Picture processing unit that renders backgrounds and sprites onto the video output module

These three components really are what the entire system boils down to. A lot of documentation out there will jump straight into the nitty-gritty details such as how the ppu handles the multiplexing of the bitmap data, or how the cpu will behave during specific read and write cycles and explain arcane and esoteric information that may not be particularly useful without a simple overview and baseline to fall back on. This Section will detail the main blocks of the NES and how they all fit together so that perhaps future endeavors will not end up wasting nearly as much time as we did trying to make sense of all the documentation out there.

2.1 Memory and Instructions

The best place to start understanding what the nes does is by looking at the end goal of the nes and looking at where all of that starts. The goal of the nes, is the same goal of any videogame, to allow the programmer/designer of the game to draw pictures for the player based on the players input. The Cartridge itself contains two major sections: the PRG-ROM and the CHR-ROM. Although they are both part of the same physical board, they do not exist on the same address space. The only the CPU has direct addressing access to the PRG-ROM while only the PPU has direct access to the data within the CHR-ROM.

What this means is that the PPU has the entire graphical library for the game already loaded and ready to go when the cartridge is plugged into the system, Which means the only thing the cpu has to do is to read its' instructions and tell the PPU which bit of bitmap data to draw rather than have to deal with loading graphical data itself.

3 PPU Rendering

The rendering system for the NES picture processing unit can be described as two different pipelines that generate two layers of graphics. A "Sprite" layer and a "Background" layer (lovingly dubbed spr, and bkg layers in our verilog code.) Video data is generated for each of these layers one pixel at a time. The pixels from either layer are then multiplexed together with a priority bit to determine whether or not the sprite should be drawn overlapping the background.

3.1 PPU memory overview

Before we continue talking about the specifics of how the NES renders the images it's important to understand the memory structure of the Picture processing unit.

Ignoring the specific addresses of the sections it's important to just understand what each of the sections do

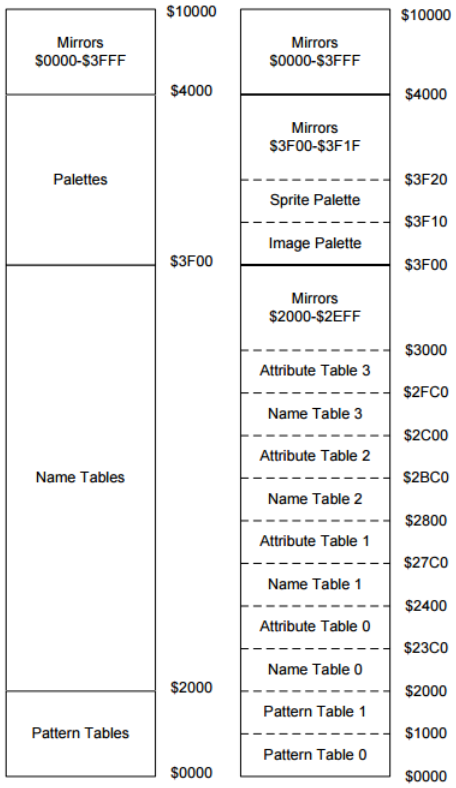


Figure 1: Nintendo NES memory management [1]

- **Object attribute memory** - RAM that stores data used in the sprite rendering pipeline
- **Name table(0-3)** - RAM that stores the per tile data used for background rendering
- **Attribute table(0-3)** - Ram paired with a respective nametable, used for decoding colour data
- **Pattern Table** - aka. CHR_ROM, this is where the raw bitmap data is stored.
- **Pallete Memory** - Split further into two 16 byte chunks, one for sprites, and one for the background

3.2 Bitmap Data storage and encoding

The guys who invented the nes had a fairly interesting method of storing bitmaps. In theory, the nes was a 6 bit colour system with 2^6 possible colours that each picture could take up. However the bitmaps themselves that make up the graphical data that exist in the game are only sixteen bytes for each 8x8 tile. The Background Rendering and Bitmap Decoding section shows a practical example of how this is actually used, but basically bitmap data in the NES's Pattern tables, are two pairs of 8 bytes store sequentially in memory. These are split int what we call "Upper" and Lower Bits. The data itself seems to be gibberish until you realize that they are meant to be stacked one on top of each other and each pair of bytes provides you with 8 pixels worth of 2 bit colour data. As can be seen in fig.2.

However this can't be right! The nes has support for 6 bit colour so it should be able to give each pixel much more colours than just two bits. And it certainly does, these two bits themselves dont produce a colour, but instead select one of four different colours. The colours that are available to each bitmap varies depending on it's rendering scheme and whether it's a Sprite or a Background tile. Because he two bits of colour are appended to two "pallete" select bits. For a total variety of 16 colours available to the program, and four colours available per bitmap. Despite these limitations a lot of graphical fidelity can still be tricked out of the system by a clever artist or programmer just look at fig.3

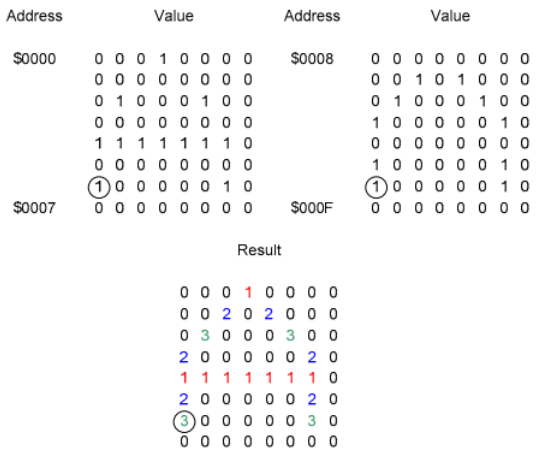


Figure 2: How the bits stack together [1]

Figure 3: CastleVania's amazing art despite limitations[1]

Another advantage to this system is that it allows the artist to have multiple "versions" of each tile that was drawn, by simply drawing it again but with a different pallete. Another important thing to note is that while the pallete for the most part is set when drawing a certain picture, the CPU and the game program is what loads the palletes with their data in the first place, and the palletes themselves are able to change from scene to scene.

3.3 Background Rendering and Bitmap decoding

The NES screen is composed of 8 pixel x 8 pixel tiles laid out to fill the screen area with 32x30 = 960 tiles. A section of the PPU memory known as the nametable contains a one byte pointer for each tile, ordered from left to right and top to bottom, for a total space in memory of 960 bytes. The pointers contained in the nametable point to the pattern table. The pattern table contains 16 bytes of information for each possible tile, 2 bytes per horizontal line. To get the information needed for an 8x1 horizontal slice, the two bytes are stacked in a sense as seen in the example below. In this example the top row represent the 16 bytes of data for the tile and the highlighted bytes represent information for the 3rd row in the tile. The bytes are combined so that the first byte represents the LSB of the colour information and the second byte represent the MSB of the colour. In this example the left-most pixel in the 3rd row is the colour 2. Clearly

Figure 4: Visual representation of the background pattern table for Donkey Kong

FC	B8	78	78	B0	78	FC	FE		FC	F8	F8	F8	F8	FC	FE	FF
			0x78		0	1	1	1		1	0	0	0			
			0xF8		1	1	1	1		1	0	0	0			
				=	10	= Colour #2										

Figure 5: Decoding of Pattern Table Data

each pixel can only be one of four colours and one of those options is always considered transparent, allowing only three actual colours in a tile. Further colours are provided by the use of the attribute table. The attribute table is a 64 byte block of memory located at the end of the nametable. The attribute table splits of the screen into 32x32 pixel tiles, each composed of 4x4 of the tiles used by the nametable. The attribute tile is split into four 2x2 tiles, and each tile is assigned a colour palette to use. The image below shows how the byte of information is used to provide colour info for the tiles. Using this scheme allows the NES to provide more colours however it still limits 2x2 tile blocks to a single palette. In our implementation on the FPGA there proved to be an issue with trying to grab both bytes of data from the pattern table on the same clock cycle. This problem was fixed by fetching tile slices one tile in advance, and splitting the fetch into two clock cycles, one for each byte. Most of the background rendering was accomplished by simply knowing the current pixels X-Y coordinate and using that information to determine which the attribute tile number, nametable tile number, row within the nametable tile, and pixel within the row was currently being rendered. An alternative would have been to use counters for each required pointer but it was felt that it would be simpler just to increment the x and y values and determine all the other information from that by using shifting or the modulo operator. In order to test the background rendering we used a static scene from the Donkey Kong game. Using an emulator we were able to pause the game and read the data from the nametable and plug that into our system. The output from our system can be seen on the left in the figure below (the colours are washed out due to taking a picture of the screen) and the original game image can be seen on the right. (The Mario image on the left is a result of a simultaneous test of the sprite rendering module.

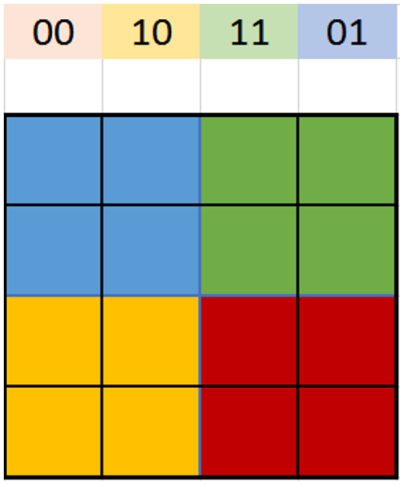


Figure 6: Decoding of Attribute Table Data

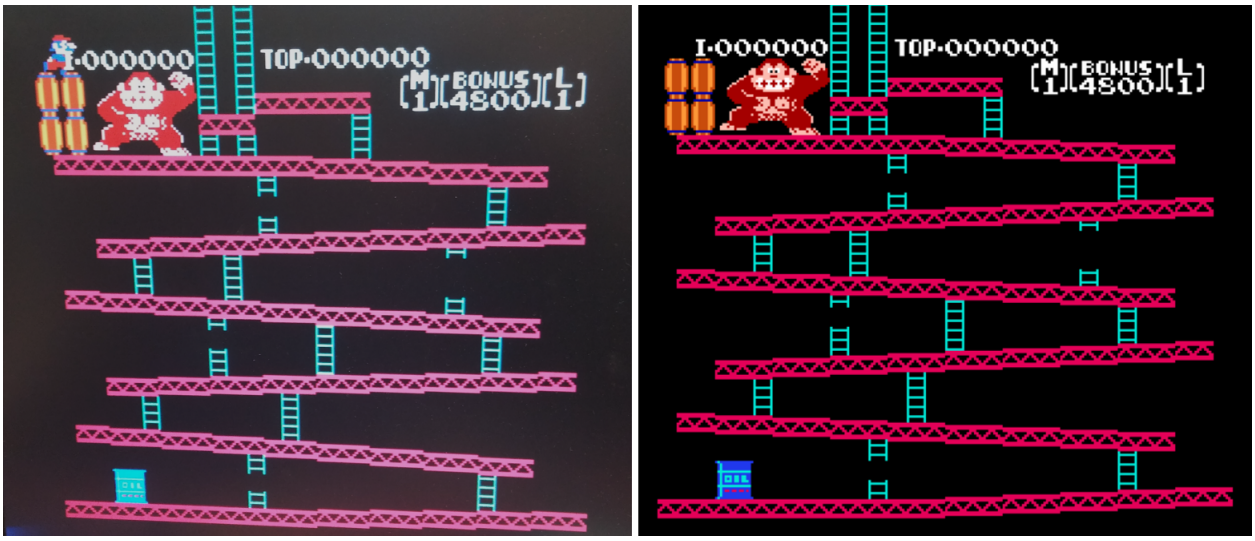


Figure 7: Comparison of original game image (Right) and our re-creation (Left)

3.4 Sprites

Sprites are very similar to background tiles in essence, they come from CHR ROM and are even stored in the exact same encoding style. However here are the key differences between Sprites and background tiles:

- Sprites can be rendered, centered and placed about any pixel (unlike tiles which have to be affixed to grids)
- Sprites are limited in number, only 64 individual sprites can possibly exist at a given time, and only 8 can exist on the same line.
- Sprites have their own separate pallete different from the background, (But it is stored in the same area of memory, works the exact same way.)

each sprite is allocated 4 bytes of memory to use to store their attribute data, (with a total of 64 sprites maximum)

- byte 1 : ypos, the y-position of the sprite on the screen

- byte 2 : spr_index, the index of the tile that represents the sprite in CHR-ROM
- byte 3 : spr_attr, the attributes of the sprite, most importantly: priority, h and v flip, pallete index
- byte 4 : xpos, the x position of the sprite on the screen

So sprite rendering is broken down into two separate operations, One that happens at the start of a scanline which evaluates which sprites should be rendered for the next scanline, and loads the relevant data into the proper renderer's buffer, this is referred to as scan in our verilog code.

The second operation which happens asynchronously from scan is called drawing, and this happens on a pixel by pixel basis.

There is also a third operation that happens only once every scanline called 'render' in our code, and this is when the rendering modules move to the next scanline by latching in the data stored in the input buffer from the 'scan' operation.

On the scan operation, a linear scan is performed searching for all sprites that would be drawn on the next scanline. if a sprite is found, to be something that needs to be drawn on the next scanline, its' corresponding data which is stored in CHR rom would be fetched and stored in one of the 8 sprite render buffers. at the end of each scanline, during the backporch, when all rendering is done for this scanline, a 'render' happens, this causes each of the spr_rend modules to latch in the data and format it in a proper way, performing flipping and priority checks on the bitmap data if necessary.

With the data latched into each of the 8 renderer modules, the draw flags are set. And each renderer knows if it will be rendering a sprite this scanline or not.

Then the drawing phase happens for each sprite renderer as the pixel_x statemachine variable runs across the scanline, each sprite render will output bitmap data so long as the current x position is within drawing range of the bitmap it holds in its current buffer.

Finally a master multiplexer multiplexes the output data from each of the eight renderers and composites the sprite data for the entire scanline to the output.

There are 8 spr_rend modules in total, Higher numbered modules take precedence over lower numbered ones, as on each scanline, during the 'scan' operation, each bitmap is loaded into the spr_rend_buf bitmap buffers from the lower numbered module first.

3.5 spr and bkg compositing

With the background renderer and the sprite renderer in place, we now have two streams of data, these two streams of video data are added together with the following compositng scheme that allows for transparency from both sprite and background

- if Sprite has priority bit high, and sprite does not have colour 00, render sprite over background
- if Sprite does not have priority bit high and background does not have a colour of 00, then render background over sprite.
- otherwise render the background pixel

This actually makes sprites a little less versitle than backgrounds in terms of colours available as one of the sprites colours b00 is always used as a transparency, this means that sprites only have 3 colours available.

This data is then fed to the VGA framebuffer and colour converter

4 VGA video Output

The FPGA is well-suited for VGA control as it allows for precise timing control that will meet the VGA standards. To display the NESs 256x240 pixel window, we used an output resolution of 640x480 at a 60 Hz refresh rate. The NES display was scaled by slowing down the VGA clock in order for one pixel to be displayed for two x periods, and each scanline was written to the VGA output twice, giving us a resolution of 512x480. The extra horizontal space was filled with a black border. The pins required for using the VGA display are red, green, blue, horizontal sync (HSYNC), and vertical sync (VSYNC). The RGB pins are analog levels ranging from 0 to 0.7V. The HSYNC and VSYNC pins operate at digital levels of 0 and 3.3V. The display works by creating one pixel at a time, scanning across one horizontal line at a time, starting with the provided RGB values for the visible area, then a front porch, an active low horizontal sync pulse, and finally a back porch. Likewise, there will be a vertical area of visible RGB values, followed by a front porch, vertical sync pulse, and back porch. When not in the visible range, RGB values must be set to zero. The basic shape of the frame is seen below, and the standard timings for 640x480 @ 60Hz follow.

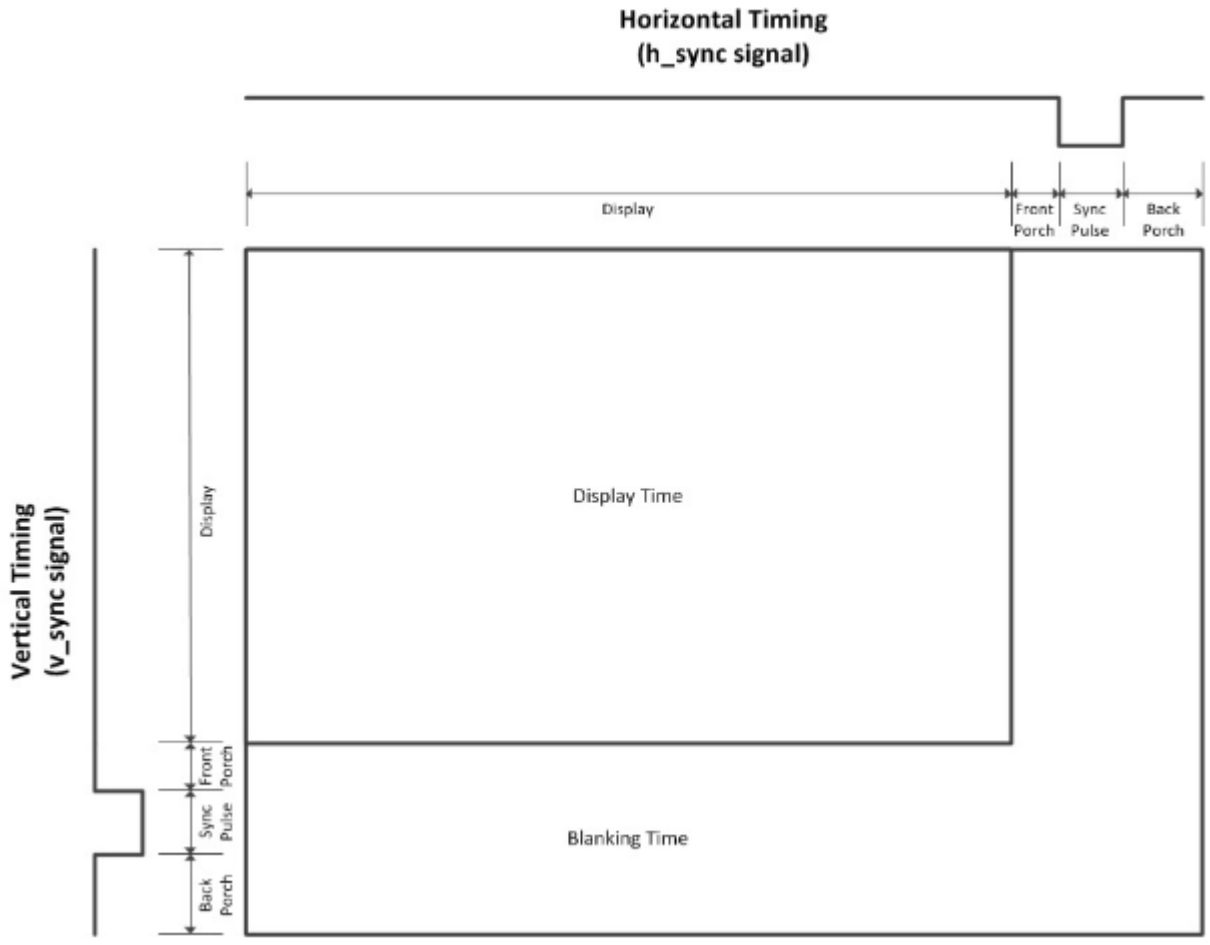


Figure 8: VGA timing diagram for h and vsyncs [2]

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Figure 9: VGA timing per scanline timings [3]

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole frame	525	16.683217477656

Figure 10: VGA timing per frame timings [3]

Although the timing requirements specify a 25.175 MHz pixel clock, this is not strictly required, as the only timing information provided to the VGA input is the horizontal and vertical sync pulses. We decided to use a 12.5 MHz clock, easily derived from the DE0 Nano 50 MHz clock to control the VGA module. This allowed the pixels to be stretched to twice their original size horizontally to easily scale the 256 pixel NES width to an equivalent of 512 pixels. The modified horizontal timing is presented in table.10.

This modified timing created a refresh rate of 59.52 Hz, a difference of less than 1% from the ideal, which proved to be compatible with all monitors we tested on. As mentioned earlier, the RGB values required analog levels from 0 to 0.7V. We decided to use 3 bits for each colour, giving us 7 non-zero levels for each colour. A simple resistor-based DAC was used to achieve these levels, as shown in the schematic fig.11. Component selections were limited to the resistors we had on hand. The VGA terminal provides 75 of resistance on each RGB pin. For instance a digital value of 011 would provide us a voltage level of

$$V_{out} = \frac{R_2||R_{vga}}{R_2||R_{vga} + R_0||R_1}(3.3V) = 0.262V$$

The VGA module has two counters that keep track of the current x-position and y-position on the screen. When a scanline starts, the RGB values are kept black to pad the missing pixels when rendering the 512 equivalent pixels on a 640-pixel

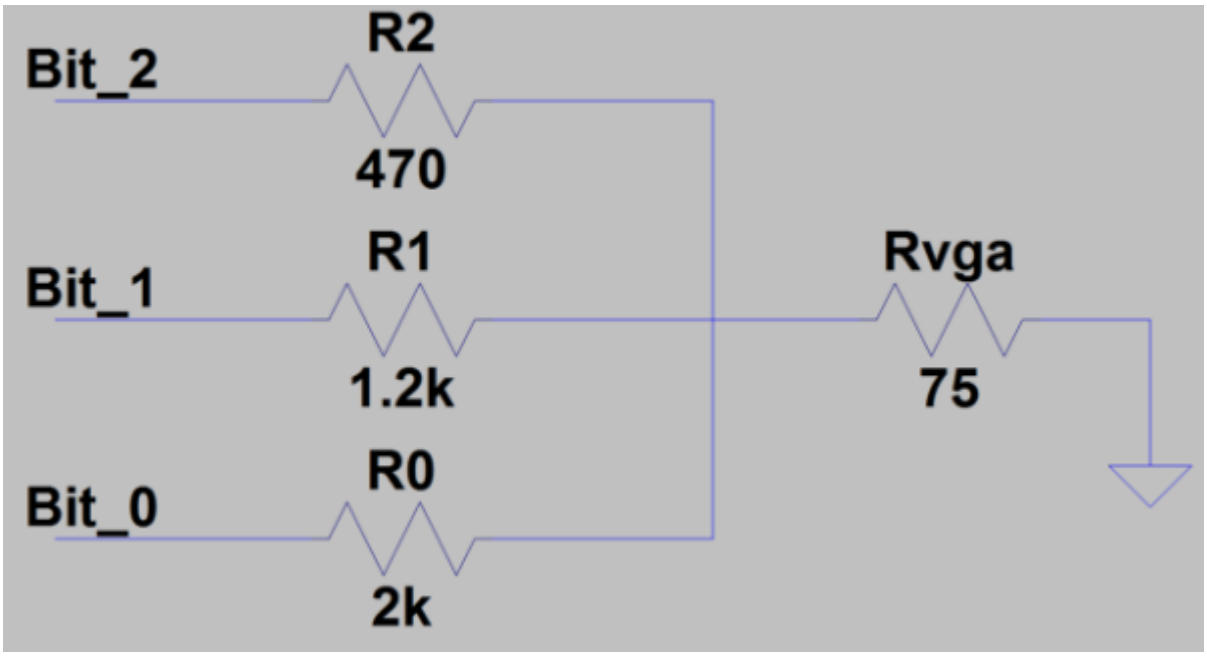


Figure 11: VGA 3-bit digital-to-analog circuit

scanline. After the padding area, the module gets RGB info from the frame buffer module by providing x and y pointers to the buffer, and this RGB info is output to the VGA pins. When the visible area is finished the RGB pins are set to zero. The HSYNC is pulsed low for a time determined by the x-counter based on the modified timing table above. After 400 clock cycles the y-counter is incremented and the x-counter is reset to zero to begin the next line. This process continues to fill the screen. The final 45 horizontal lines are all blank to provide the correct timing for the front porch, vertical sync pulse, and back porch needed in the vertical timing.

4.1 NES colour conversion

The old school genuine colours for the nes are referenced by colourcodes that correspond to a specific colour, when converted into standard html colours they are seen in figure 13

Code	preconverted	conversion
0	#757575	#6D6D6D
1	#271B8F	#242492
2	#0000AB	#0000B6
3	#47009F	#490092
4	#8F0077	#92006D
5	#AB0013	#B60000
6	#A70000	#B60000
7	#7F0B00	#6D0000
8	#432F00	#492400
9	#004700	#004900

Figure 12: NES colours converted to html

With a simple conversion to 9 bit colour(3 bits for each of the colours, details can be seen in appendix B), we were able to confirm that the colours match up very closely and very nicely.

Code	preconverted
0	#757575
1	#271B8F
2	#0000AB
3	#47009F
4	#8F0077
5	#AB0013
6	#A70000
7	#7F0B00
8	#432F00
9	#004700

Figure 13: NES colours converted to html

one more quick conversion can be done to change the colours into parallel output data.

conversion	R	G	B	Binary Output		
#6D6D6D	3	3	3	011	011	011
#242492	1	1	4	001	001	100
#0000B6	0	0	5	000	000	101
#490092	2	0	4	010	000	100
#92006D	4	0	3	100	000	011
#B60000	5	0	1	101	000	001
#B60000	5	0	0	101	000	000
#6D0000	3	0	0	011	000	000

Figure 14: NES colours converted to html

5 Conclusion

This was a huge project and had taken a lot of time to complete. Over the course of this project we learned a lot of things about how larger systems are integrated as well as how quartus instantiates and deals with memory constructs. A lot of missteps were made along the way but the end result is a working picture processing unit that was able to display pictures and even update on the fly against real genuine nes memory.

In the future all we would have to do is create a small interface to provide the 6502 CPU to read and write data to the ppu in the way that it expects to and from there it would not be a far cry to having a working NES system on an fpga.

6 Appendix A - Code Listings

```
// Sprite render buffer
/*
        decodes the input and outputs a proper 4 bit colour pixel based on the position of a single byte
module spr_rend( clk, [31:0]rend_buf, pixel_x, rend_now, draw_now,
        output logic [3:0] pallete_colour // 4 bit output to pallete
        output logic valid
);
*/

module spr_rend(
        input logic clk,
        input logic [31:0]rend_buf,
        input logic [7:0]pixel_x,
        input logic rend_now,         // render the image by copying read buf
        input logic draw,                 // connected to spr_rend_draw_flag
        output logic [3:0] pallete_colour, // 4 bit output to pallete
        output logic valid // output is only considered for drawing by the multiplexer if draw is valid
);

parameter XPOS_LSB = 16;
parameter XPOS_MSB = 23;
parameter BIT_HFLIP = 30;
parameter BIT_PRIORITY = 29;
parameter PS_LSB = 16;
parameter PS_MSB = 17;

integer i;// index for for reversing bits
logic [1:0] pallete = 0;
logic [1:0] bmp_output;
logic [15:0] bmp_data;
logic bkg_priority;
logic hflip;
logic [2:0] vslice =0;
logic [7:0] xpos;
logic [1:0] bmp[7:0];

assign pallete_colour = {pallete, bmp_output};

always_ff@(posedge rend_now)begin
        pallete = {rend_buf[PS_MSB],rend_buf[PS_LSB]};
        if(!(hflip)) begin
                for (i  = 0 ; i < 8; i++ )begin
                        // LSB
                        bmp_data[i] = rend_buf[7 - i];
                        // MSB
                        bmp_data[i+8] = rend_buf[15 - i];
                end
        end else begin
                bmp_data = rend_buf[15:0];
        end
        bkg_priority = rend_buf[BIT_PRIORITY];
```

```
52          hflip = rend_buf[BIT_HFLIP];
53          xpos = rend_buf[XPOS_MSB:XPOS_LSB];
54  end
55
56  always_ff@(negedge clk)begin
57          if((pixel_x >= xpos) && (pixel_x < xpos + 8) && draw) begin
58                  valid = 1;
59                  vslice = pixel_x - xpos;
60                  bmp_output = {bmp_data[vslice+8], bmp_data[vslice]};
61          end
62          else valid = 0;
63
64  end
65
66  endmodule
```

```
1   // Vga stream module for use with an SDRAM dma which is currently unfinished
2   module vga_stream(
3           input logic cpu_latch,                                  //
4           input logic reset,                                  // .reset
5           input logic cpu_write,                                  // .cpu_ckin
6           input logic [5:0]c_code_cpu,  // .data_stream_in
7           input logic vga_clk,                                      // .vga_clk
8           // === Conduit ====
9           output logic [8:0]rgb_coe,          // .conduit
10          output logic vsync,                                     //
11          output logic hsync,                                     //
12          output logic vga_read,                          // On reading = high, req next pixel
13          output logic done
14  ); // VGA avalon interface,
15
16
17  logic [5:0]scanline[255:0];
18  logic [8:0]coloursDecode[63:0];
19  logic [8:0]rgb_buf;
20  logic [5:0]c_code_vga;
21  logic [7:0]pix_ptr_x;
22  logic [7:0]pix_ptr_y;
23
24  always_ff@(posedge(vga_clk)) begin
25          if(vga_read)
26                  scanline[pix_ptr_x] = c_code_vga;
27          rgb_buf = coloursDecode[scanline[pix_ptr_x]];
28  end
29
30  initial begin
31          $readmemh("vga_colours_rgb.txt",coloursDecode);
32  end
33
34  vga_out vga_0(
35          .pix_clk(vga_clk), .pix_ptr_x, .pix_ptr_y,.rgb(rgb_coe),
36          .vsync, .hsync, .reading(vga_read), .rgb_buf
37  );
38
```

```
39 nes_video_dc_fifo dc_fifo(
40         .din(c_code_cpu),
41         .clk_write(cpu_latch),
42         .read(vga_read),
43         .write(cpu_write),
44         .clk_read(vga_clk),
45         .dout(c_code_vga),
46         .reset,
47         .done
48 );
49
50 endmodule
51
52
53 module nes_video_dc_fifo (// This thing does NOT check for errors
54         input logic [31:0] din,
55         input logic clk_write,
56         input logic read,
57         input logic write,
58         input logic clk_read,
59         output logic [5:0]dout,
60         input logic reset,
61         output logic done
62 );
63
64 logic [5:0] vid_fifo[512:0]; // size be a bit overkill
65 logic ptr_read = 0;
66 logic ptr_write = 0;
67
68 assign done = (ptr_write == ptr_read);
69 // if write is high ptr will increment and data will be pushed in
70 always_ff@(posedge clk_write) begin
71         if(reset) begin
72                 ptr_write = 0;
73         end else begin
74                 if(write)begin
75                         vid_fifo[ptr_write] = {din[5:0]};
76                         ptr_write = ptr_write + 1;
77                 end
78         end
79 end
80 // if read is high, ptr will increment and data will be pushed out
81 always_ff@(posedge clk_read) begin
82         if(reset) begin
83                 ptr_read = 0;
84         end else begin
85                 if(read&&(!done))begin
86                         dout = vid_fifo[ptr_read];
87                         if(!done)
88                                 ptr_read = ptr_read + 1;
89                 end
90         end
91 end
92
```

```
93 endmodule


1  // VGA ROM top level test.
2
3  // Tests the following:
4  // VGA output operation - working but glitchy
5  // NES colour decoding - passed
6
7  // Right now we need to solve the "square problem"
8  // some anomalies are observed on the right side of the screen
9
10
11 // This thing also tests uart_port TX and RX loopback
12
13 // (PPU REPLACED BY TEST_IMG_DUMMY_ROM) ->
14 module vga_rom_test_top(
15         input logic uart_port_DI,// gpio 31
16         output logic uart_port_DO, // gpio 33
17         input logic CLOCK_50, rst,
18         output logic [2:0]RED,
19         output logic [2:0]GREEN,
20         output logic [2:0]BLUE,
21         output logic HSYNC, VSYNC);
22
23         // ====== RX signals ======
24                 logic [15:0]read_ptr; // buffer read pointer
25                 logic rx_clear;
26                 logic read_valid;
27                 logic [7:0]uart_DO;
28                 //===== tx signals =======
29                 logic[15:0]send_ptr;
30                 logic tx_clear;
31                 logic [7:0]tx_DI;
32                 logic send_done;
33
34
35         uart_port uart_0(.clk(ppu_clk),.*);
36
37         logic nios_clk;
38         logic pix_clk;
39         logic ppu_clk;
40         logic ppu_slow_clk;
41         logic A203_clk;
42
43         logic [5:0]test_read_col[240];
44
45         logic [7:0]fb_ptr_x;
46         logic [7:0]fb_ptr_y;
47
48         logic [7:0]ppu_ptr_x;
49         logic [7:0]ppu_ptr_y;
50         logic [8:0]rgb;
51         logic [5:0]testColours;
52
```

```
53          // PIN outputs
54          logic vsync;
55          logic hsync;
56          logic [8:0]rgb_OUT;
57
58          assign RED = rgb_OUT[8:6];
59          assign GREEN = rgb_OUT[5:3];
60          assign BLUE = rgb_OUT[2:0];
61          assign VSYNC = vsync;
62          assign HSYNC = hsync;
63
64
65          // Test variables
66          integer i, j, k;
67          logic [7:0]grid_center;// rolls over at 128
68          logic [7:0]h_line = 120;// rolls over at 128
69          logic [7:0]y_line = 0;// rolls over at 128
70          logic [7:0]h_line_values[3:0];
71          logic [7:0]y_line_values[3:0];
72          integer n_line = 0;
73          integer progState;
74
75          assign h_line = grid_center;
76          assign y_line = grid_center;
77          initial begin
78                  h_line_values[0] = 0;
79                  h_line_values[1] = 255 - 10;
80                  h_line_values[2] = 255 - 10;
81                  h_line_values[3] = 0;
82
83                  y_line_values[0] = 0;
84                  y_line_values[1] = 000;
85                  y_line_values[2] = 239 - 10;
86                  y_line_values[3] = 239 - 10;
87          end
88          always_ff@(posedge ppu_clk) begin //
89                  if(progState < 256) begin // state 0 = reset
90                          i = 0;
91                          progState = progState + 1;
92                  end else begin
93                          if(i < (240*256 + 1)) begin
94                                  if(ppu_ptr_x == 255)
95                                          ppu_ptr_y = ppu_ptr_y +1;
96                                  ppu_ptr_x = ppu_ptr_x+1;
97
98                                  if(ppu_ptr_x <y_line)begin
99                                          if(ppu_ptr_y <h_line)
100                                                 testColours = 'h27; // Orange
101                                         else
102                                                 testColours = 'h14;// magenta
103                                 end else begin
104                                         if(ppu_ptr_y <h_line)
105                                                 testColours = 'h01;// blue
106                                         else
```

```
                                                testColours = 'h2b;// green
                                end

                                i = i + 1;
                        end else i = 0; // Draw again
                        // Every 1/12th of a second, move the grid_center
                        // diagonally
                        if(k < 1200000) k = k +1;
                        else begin
                                k = 0;
                                grid_center = grid_center+1;
                        end
                end
        end

        // VGA output initialization
        vga_out vgao_dut(
                .pix_clk(pix_clk), .rgb_buf(rgb),
                .pix_ptr_x(fb_ptr_x), .pix_ptr_y(fb_ptr_y),
                .rgb(rgb_OUT),. vsync(vsync), .hsync(hsync)
                );

        // frame buffer initialization
        vga_fb fb_dut(
                .ppu_ptr_x(ppu_ptr_x), .ppu_ptr_y(ppu_ptr_y),
                .ppu_ctl_clk(ppu_clk), .CS(1),
                .ppu_DI(testColours),
                .pix_ptr_x(fb_ptr_x), .pix_ptr_y(fb_ptr_y),
                .rgb(rgb), .pix_clk(pix_clk)
                );

        initial begin
                progState = 0;
                $readmemh("pixtest.txt", test_read_col);
        end

        // initialize clocks
        clocks        clock_inst (
        .inclk0 ( CLOCK_50 ), // 50Mhz input
        .c0 ( nios_clk ),                        // 50Mhz Output Phase locked
        .c1 ( ppu_clk ),                        // 21.428 Mhz ppu clock
        .c2 ( ppu_slow_clk ),                        // ppu div 4 5.35 mhz clk
        .c3 ( A203_clk ),                        // 1.785 Mhz 6502 clock
        .c4 ( pix_clk )                        // 12.5Mhz vga Clock
        );

endmodule


// vga_fb.sv test bench
// Peter li


module vga_fb_tb();
        // inputs
```

```
        logic ppu_ctl_clk;
        logic [7:0]ppu_ptr_x;
        logic [7:0]ppu_ptr_y;
        logic [5:0]ppu_DI;
        logic CS;
        logic [7:0]pix_ptr_x;
        logic [7:0]pix_ptr_y;
        //output
        logic [8:0]rgb;
        vga_fb dut(.pix_clk(ppu_ctl_clk),.*);

        integer i,j,k,l;
        initial begin
                CS = 1;
                ppu_ctl_clk = 0;
                ppu_ptr_x = 0;
                ppu_ptr_y = 23;
                pix_ptr_x = 0;
                pix_ptr_y = 23;
                ppu_DI = 03; // Should write 010 000 100 to rgb
                #80ns;
                for (i =0; i< 240; i++)begin
                        for (j = 0; j<256; j++) begin
                        pix_ptr_x = j-1; ppu_ptr_x = j;
                        pix_ptr_y = i; ppu_ptr_y = i;
                        ppu_DI = i+j*6;
                        #40ns;
                        $display("p: %d %d code: %x rgb : %x ",pix_ptr_x, pix_ptr_y,ppu_DI,rgb);
                        end
                end
                $stop;
        end

        always begin
                #20ns;
                ppu_ctl_clk = ~ppu_ctl_clk;
        end
endmodule


// uart.sv - Peter Li ( this module is not used in the current project )

/*
        simple uart module used as a general purpose Comms channel with the FPGA

        use USB to FTDI Card.

        asynch_in
        synch_out

        settings for com port
        115200 baud
        1 start bit
        no parity
        1 stop bit (or more, it doesnt matter on RX side, tx will output 1)
```

```
16          8 bit data
17
18          --- How to use ---
19
20          - instantiate uart_port()
21          simply connect your UART signal to uart-port-DI
22
23          tx data can be sent by writing to the buffer then incrementing the send_ptr,
24
25          available data will be present in the 64kB buffer, this is dereferenced via the read_ptr
26
27          to prevent buffer from overflowing hold CLEAR signal high
28          to read after X number of bits are available set read_ptr
29          to X and wait for read_valid
30  */
31
32
33  module uart_port( // instantiates the entire port
34                  input logic clk,
35                  // ====== RX signals ======
36                  input logic [15:0]read_ptr, // buffer read pointer
37                  input logic rx_clear,
38                  output logic read_valid,
39                  output logic [7:0]uart_DO,
40                  //===== tx signals =======
41                  input logic[15:0]send_ptr,
42                  input logic tx_clear,
43                  input logic [7:0]tx_DI,
44                  output logic send_done,
45                  // ===== Phyiscal output pins =======
46                  input logic uart_port_DI,
47                  output logic uart_port_DO
48          );
49          logic uart_valid;
50          logic [7:0]uart_rx_DO;
51          uart_tx uart_transmitter(
52                  .send_ptr,
53                  .uart_port_DO,
54                  .clear(tx_clear),
55                  .tx_DI,
56                  .send_done,
57                  .clk
58          );
59
60          uart_buf uart_buffer(
61                  .read_ptr, .uart_DI(uart_rx_DO),
62                  .read_clk(clk),
63                  .uart_valid,
64                  .clear(rx_clear),
65                  .uart_DO(uart_DO),
66                  .read_valid
67          );
68          uart_rx uart_reciever(
69                  .uart_DI(uart_port_DI),
```

```
70                  .clk,
71                  .uart_valid,
72                  .uart_DO(uart_rx_DO)
73          );
74  endmodule
75
76  module uart_buf(//  64kB UART read buffer
77          input logic [15:0]read_ptr,
78          input logic [7:0]uart_DI,
79          input logic uart_valid, read_clk,
80          input logic clear, // if purge is pulled high, the rx_ptr will go to 0
81          output logic[7:0]uart_DO,
82          output logic read_valid
83  );
84          logic [15:0]rx_ptr = 0;
85          logic [15:0]rx_ptr_next = 0;
86          logic [7:0]rx_buf['h600f:0];
87          // On each read_clk data is read out to the controling module
88          always_ff@(posedge read_clk) begin
89                  uart_DO = rx_buf[read_ptr];
90                  if(clear) begin
91                          rx_ptr = 0;
92                  end else begin
93                          rx_ptr = rx_ptr_next;
94                  end
95          end
96
97          assign read_valid = (rx_ptr >= read_ptr );
98          // data is valid so long as the rx_ptr also if the buffer is full then all data locations are valid
99          // is greater than the requested data location
100         // Data is read into the current ptr location and the pointer is incremented
101         always_ff@(posedge uart_valid)begin
102                 rx_buf[rx_ptr] = uart_DI;
103                 rx_ptr_next = rx_ptr + 1;
104         end
105 endmodule
106
107
108 module uart_rx( // uart in and parallel 8 bit out
109         input logic uart_DI,
110         input logic clk,        // operates at will borrow a 21.477 mhz clk borrowed from the ppu_clk
111         output logic uart_valid,
112         output logic [7:0]uart_DO
113 );
114         parameter NCLKS_PER_BIT = 186; // 21 477 000/(115200) ~= 186.4 Cycles of oversampling
115
116         // Depending on FTDI settings this could be different
117         parameter SPACE = 1;
118         parameter MARK = 0;
119
120         parameter IDLE = SPACE; // IDLE is always space
121         parameter START = !IDLE; // start is always the opposite of space
122         parameter STOP = !START; // stop bit is always the opposite of start
123
```

```
124         // RX machine states
125         parameter WAITING = 0;
126         parameter READING_DI = 1;
127         parameter STOPPING = 2;
128         parameter START_BIT = 3;
129         logic [2:0]bit_ptr = 0;
130         logic [1:0]state = WAITING;
131         logic [15:0]count = 0;
132
133         // RX fsm
134         always_ff@(posedge clk) begin
135                 case(state)
136                         WAITING: begin
137                                 uart_valid <= 0;
138                                 if(uart_DI == START) begin
139                                         state <= START_BIT;                 // proceed to start bit
140                                         count <= 0;                                         // Start the cou
141                                 end
142                         end
143                         START_BIT: begin
144                                 if(count == NCLKS_PER_BIT * 276) begin
145                                         count <= 0;
146                                         bit_ptr <= bit_ptr + 1;
147                                         uart_DO[bit_ptr] <= uart_DI ^ SPACE;
148                                         state <= READING_DI;
149                                 end else begin
150                                         count <= count + 1;
151                                 end
152                         end
153                         READING_DI: begin
154                                 if(count == NCLKS_PER_BIT) begin
155                                         count <= 0;
156                                         bit_ptr <= bit_ptr + 1;
157                                         uart_DO[bit_ptr] <= uart_DI ^ SPACE;
158                                         if(bit_ptr == 7) state <= STOPPING;
159                                 end else begin
160                                         count <= count + 1;
161                                 end
162                         end
163
164                         STOPPING: begin
165                                 bit_ptr <= 0; // Should already be at zero but this will ensure
166                                 if(count > NCLKS_PER_BIT/2)
167                                         uart_valid <= 1;
168
169                                 if(count == NCLKS_PER_BIT)begin
170                                         count <=0;
171                                         state <= WAITING;
172                                 end else begin
173                                         count <= count + 1;
174                                 end
175                         end
176
177                 endcase
```

```
178         end
179 endmodule
180
181 module uart_tx(
182         input logic clk,clear, // Same clock as RX
183         input logic [7:0]tx_DI,
184         input logic [15:0]send_ptr,
185         output logic uart_port_DO,
186         output logic send_done
187 );
188
189         parameter NCLKS_PER_BIT = 186; // 21 477 000/(115200) ~=
190         parameter NCLKS_PER_BIT_1_5 = 276;
191         //186.4 Cycles of oversampling
192
193         // Depending on FTDI settings this could be different
194         parameter SPACE = 1;
195         parameter MARK = 0;
196
197         parameter IDLE = SPACE; // IDLE is always space
198         parameter START = !IDLE; // start is always the opposite of
199         //space
200         parameter STOP = !START; // stop bit is always the opposite of start
201
202         parameter WAITING = 0;
203         parameter SENDING_DO = 1;
204         parameter STOPPING = 2;
205         parameter START_BIT = 3;
206
207         logic [7:0]tx_buf[7:0];        // 8 byte Output buffer
208         logic [7:0]tx_ptr = 0;                                  // data control pointer
209         logic [2:0]tx_bit_ptr =0;                          // pointer for bitwise send
210         logic [15:0]count = 0;                                  // Timing Counter for
211         logic [1:0]state = WAITING;                                 // state for the FSM
212
213         // Per byte tx logic (fsm)
214         always_ff@(posedge clk )begin
215                 tx_buf[send_ptr] = tx_DI;
216                 send_done = (tx_ptr >= send_ptr); //data in buffer isnt considered valid unless the send po
217                 if(clear)begin
218                         tx_ptr = 0;
219                 end
220                 case(state)
221                         WAITING: begin
222                                 uart_port_DO <= SPACE;
223                                 // The user will increment
224                                 // send_ptr as soon as there
225                                 // is data
226                                 if(send_ptr > tx_ptr) begin
227                                         count <= 0;
228                                         state <= START_BIT;
229                                 end
230                         end
231
```

```
                        START_BIT: begin // Send the start bit
                                uart_port_DO <= MARK;
                                tx_bit_ptr = 0;
                                if(count == NCLKS_PER_BIT)begin
                                        count <= 0;
                                        state <= SENDING_DO;
                                end else begin
                                        count <= count+1;
                                end
                        end
                        SENDING_DO: begin
                                uart_port_DO <= SPACE^tx_buf[tx_ptr][tx_bit_ptr];
                                if(count == NCLKS_PER_BIT) begin
                                        count <= 0;
                                        tx_bit_ptr <= tx_bit_ptr + 1;
                                        if(tx_bit_ptr ==7) state <= STOPPING;
                                end else begin
                                        count <= count + 1;
                                end
                        end
                        STOPPING: begin         // send the stop bit and end bytestream
                                uart_port_DO <= STOP;
                                tx_bit_ptr <= 0;
                                tx_ptr <= tx_ptr + 1;
                                if(count == NCLKS_PER_BIT) begin
                                        count <=0;
                                        state <= WAITING;
                                end else begin
                                        count <= count + 1;
                                end
                        end
                endcase
        end

endmodule
```

```
/*
        uart_rx_tb.sv

        tests functionality of the uart_rx module
        Though tbh this is more of a rx/tx module

    Author: Peter Li
*/

module uart_rx_tb();
        parameter baud = 115200;
        // INPUTS
        logic [15:0]read_ptr = 26; // buffer read pointer
        logic clk = 0;
        logic uart_clk = 0;
        logic rx_clear = 1;  // control signals
        logic uart_port_DI = 1; // Read UART from pin

```

```
19          logic [15:0]send_ptr=0;
20          logic tx_clear =1;
21          logic [7:0]tx_DI=0;
22          logic send_done;
23          logic uart_port_DO;
24
25          // OUTPUTS
26          logic [7:0]uart_DO;
27          logic read_valid;
28
29          logic [7:0]ascii_char = 65; // ascii char 'a'
30          // test vars
31
32          integer i, j, k, l;
33
34
35          uart_port dut(.*);
36          initial begin
37                  clk = 0;
38                  rx_clear = 0;
39
40                  /*while(!read_valid)begin
41                          #8.68us; // Send Start Bit
42                          uart_port_DI = 0;
43                          for(i = 0; i < 8; i++)begin
44                          #8.68us;
45                          uart_port_DI = !ascii_char[i];
46                          end
47                          #8.68us; // Send Stop Bit
48                          uart_port_DI = 1;
49                          ascii_char = ascii_char + 1;
50                  end
51                  */
52                  // END OF UARx test
53                  ascii_char = 'h41;//reset back to A
54                  tx_clear = 0;
55                  tx_DI = ascii_char;
56                  ascii_char = ascii_char + 1;
57                  #46.2ns;
58                  send_ptr = send_ptr + 1;
59                  tx_DI = ascii_char;
60                  #46.2ns;
61                  send_ptr = send_ptr + 1;
62                  tx_clear = 1;
63                  send_ptr = 1; // blast last transmission.
64                  #86.8us
65                  // END OF UATx test
66                          $stop;
67          end
68
69          always begin
70                  #4.34us;
71                  uart_clk = ~uart_clk;
72          end
```

```
        always begin
                #23.2ns;
                clk = ~clk;
        end


endmodule
```

```
// VGA driver, takes a 256x240 image and scales it to 640x480@60Hz VGA
// by using a 12.5MHz pixel clock, and incorporating black borders on either side

module vga_out(
        input logic pix_clk,        // 12.5 MHz clock signal

        input logic [8:0] rgb_buf,        // connect to rgb output of buffer
        output logic [7:0]pix_ptr_x,
        output logic [7:0]pix_ptr_y,

        output logic [8:0] rgb,        // 3 bits each for red, green, blue
        output logic vsync,                // vertical syncing signal, active low
        output logic hsync            // horizonal syncing signal, active low
);
        // 0-31 black
        // 32-287 NES image
        // 288-319 black
        // 320-327 front porch
        // 328-375 sync (47 cycles)
        // 376-399 back porch

        parameter NES_WIDTH = 256;
        parameter NES_HEIGHT = 240;

        // frame constants always check >= and <
        parameter L_BLANK = 0;
        parameter NES_W = 32;
        parameter R_BLANK = 288;
        parameter HF_PORCH = 320;
        parameter HSYNC_START = 328;
        parameter HB_PORCH = 376;
        parameter H_END = 400;

        parameter V_VISIBLE = 0;
        parameter VF_PORCH = 480;
        parameter VSYNC_START = 490;
        parameter VB_PORCH = 492;
        parameter V_END = 525;

        logic [9:0] pixel_x = 0;
        logic [9:0]        pixel_y = 0;

        initial begin
                // reset pixel counters
                //pixel_x = '0;
//                pixel_y = '0;
```

```
47  //                  rgb = '0;
48          end
49          always_comb begin
50                  rgb = (pixel_x >= NES_W && pixel_x < R_BLANK && pixel_y < VF_PORCH) ? rgb_buf : 0;
51                  if(pixel_x >= R_BLANK) rgb = 0;
52          end
53          always_ff @(posedge pix_clk) begin
54
55                  // RGB control
56                  //if ( pixel_x >= L_BLANK && pixel_x < NES_W)
57                  //        rgb <= '0;
58                  //if (pixel_x >= NES_W && pixel_x < R_BLANK && pixel_y < VF_PORCH) begin
59                          // RGB gets NES info // Get the information from the framebuffer module
60                  //        rgb = rgb_buf;
61                  //end
62                  //if (pixel_x >= R_BLANK)
63                  //        rgb = '0;
64
65                  // HSYNC Control
66                  if (pixel_x >= HSYNC_START && pixel_x < HB_PORCH)
67                          hsync <= 0;
68                  else
69                          hsync <= 1;
70
71                  // VSYNC Control
72                  if (pixel_y >= VSYNC_START && pixel_y < VB_PORCH)
73                          vsync <= 0;
74                  else
75                          vsync <= 1;
76
77                  // move to next pixel
78                  if (pixel_x == 10'd399) begin
79                          // reset x value
80                          pixel_x <= '0;
81
82                          // increment or reset y value
83                          if (pixel_y == 10'd524)
84                                  pixel_y <= '0;
85                          else
86                                  pixel_y <= pixel_y + 1'b1;
87                  end
88                  else
89                          pixel_x <= pixel_x + 1'b1;
90          end
91
92          always_comb begin
93                  if (pixel_x >= NES_W && pixel_x < R_BLANK)        // before or after visible area
94                          pix_ptr_x = pixel_x - NES_W;
95                  else
96                          pix_ptr_x = 0;                // set pointer for next pixel to be rendered
97
98                  // lines are doubled to fill the screen
99                  if (pixel_y < VF_PORCH)
100                         pix_ptr_y = pixel_y >> 1;        // right-shift will duplicate lines
```

```
101                 else
102                         pix_ptr_y = '0;
103         end
104 endmodule
```

```
1  DB
2  4C
3  5
4  84
5  103
6  141
7  140
8  C0
9  88
10 10
11 10
12 11
13 53
14 00
15 00
16 00
17 16D
18 1F
19 57
20 107
21 145
22 183
23 188
24 190
25 118
26 20
27 28
28 22
29 24
30 00
31 00
32 00
33 1FF
34 AF
35 E7
36 167
37 1DF
38 1DD
39 1DB
40 1E2
41 1EA
42 131
43 B2
44 BC
45 36
46 00
47 00
48 00
49 1FF
```

```
177
177
1B7
1EF
1EE
1ED
1F5
1F4
1BC
17D
17E
13F
00
00
00
```

```
// Vga_fb.sv
// Author: Peter Li, Cameron Morgan

// This is the frame buffer for vga, The frame buffer in this case is basically
// ram within the ppu It will be instantiated alongside not within the vga

/*
        Frame buffer relationships:
 +--------------+   c_codes   +----------+
 |              |   (eg. 18   |          |
 |              | +---------> |  vga     |
 |    ppu       |   gives a   |  Frame   |
 |              |   brownish  |  Buffer  |
 |              |   colour)   |          |
 |              |             +----------+
 +--------------+                  |rgb values
                                   |RRRGGGBBB
                                   v(eg. 000111000
         +--------------+           gives pure green)
         |              |     +----------+
         |   Video      |     |          |
         |              |     |  vga     |
         |              |     |  timing..|
         |              | <---+  logic.. |
         |              |     |  etc..   |
         |              |     +----------+
         +--------------+


        frame buffer is a 2dimensional array read from it with
        pix_ptr_x and pix_ptr_y

        Top left Corner is  0,0
        bottom right corner is 255,239
*/

module vga_fb(
        // Input control lines from ppu
```

```
39          input logic ppu_ctl_clk,
40          input logic [7:0]ppu_ptr_x,
41          input logic [7:0]ppu_ptr_y,
42          // TODO The ppu may not write each byte individually
43          // Investigate this
44          input logic [5:0]ppu_DI,
45          input logic CS,
46          // Output to vga_out Module
47          input logic pix_clk,
48          input logic [7:0]pix_ptr_x,
49          input logic [7:0]pix_ptr_y,
50          output logic [8:0]rgb // format of RRRGGGBBB r is always gonna be msb
51  );
52  //        logic [5:0]pixel_code[239:0][255:0];        // Frame buffer RAM
53
54          logic [5:0]pix;
55          logic [2:0]r;
56          logic [2:0]g;
57          logic [2:0]b;
58          logic [8:0]coloursDecode[63:0];
59          logic [8:0]dec;
60
61          logic [7:0]pix_ptr_y_clamp;
62          logic [7:0]ppu_ptr_y_clamp;
63
64          assign pix_ptr_y_clamp = pix_ptr_y > 239 ? 239: pix_ptr_y;
65          assign ppu_ptr_y_clamp = ppu_ptr_y > 239 ? 239: ppu_ptr_y;
66  sc_dpram (
67          pix_clk,
68          ppu_DI,
69          {pix_ptr_y_clamp, pix_ptr_x}, // read address
70          {ppu_ptr_y_clamp, ppu_ptr_x}, //write address
71          'h1,
72          pix);
73
74
75          initial begin
76                  $readmemh("vga_colours_rgb.txt",coloursDecode);
77          end
78
79          /*always_ff@(posedge pix_clk) begin
80           pix = pixel_code[ppu_ptr_y_clamp][ppu_ptr_x];
81          end
82
83
84          // PPU access (Write only)
85          always_ff@(posedge pix_clk) begin
86                          pixel_code[ppu_ptr_y_clamp][ppu_ptr_x] = ppu_ptr_x > 127?'h27:'h15;
87          end */
88          assign dec = coloursDecode[pix];
89          // vga out access (Read only)
90          assign rgb = {dec[8:6],dec[5:3],dec[2:0]};
91
92  endmodule
```

```
/*
        PPU.sv
    Peter Li
*/

// This version of the module tests the PPU's ability to draw images without external input
// OAM and NT will be initialized to known values
// Test image: draw the mario sprite as a background and we are gonna use NT_1 for
//

module ppu_tb(); // Static general purpose testbench for viewing internal signals

        logic [2:0] CPUA;               // PPU register select Selects ppu register 0-7 (mapped to £2000-£
        logic [7:0] CPUDI;          // CPU data input
        logic[7:0] CPUDO;           // CPU data read
        logic CPUCLK;                       // Cpu clock for read/write
        logic RW;                           // Read/Write
        logic CS;                           // Chip Select
        logic RST;                          // Chip reset
        logic NMI;                          // Non interruptable Interrupted (signifies the start of
        logic ALE;                           // Address latch enable
        logic [13:0] APPU;          // Address and data pins
        logic [7:0] PPUDO;          // PPU data output
        logic [7:0] PPUDI;          // PPU data input
        logic [5:0]VGA_STREAM_DATA; // PPU video pipeline out
        logic [7:0] PPU_PTR_X;
        logic [7:0] PPU_PTR_Y;
        logic VGA_STREAM_READY;         // ppu video ready output


logic PPU_SLOW_CLOCK=0;
always begin
        #8ns;
        PPU_SLOW_CLOCK = ~PPU_SLOW_CLOCK;
end

initial begin
        #8000us;
h        $stop;
end
ppu_core ppu(.*);

endmodule

module ppu_core(                                    // PPU Component
        input logic [2:0] CPUA,                 // PPU register select Selects ppu register 0-7 (mapped to $
        input logic [7:0] CPUDI,            // CPU data input
        output logic[7:0] CPUDO,            // CPU data read
        input logic CPUCLK,                     // Cpu clock for read/write
        input logic RW,                         // Read/Write
        input logic CS,                         // Chip Select
        input logic RST,                        // Chip reset
        output logic NMI,                        // Non interruptable Interrupted (signifies the st
        output logic ALE,                       // Address latch enable
```

```
55          output logic [13:0] APPU,          // Address and data pins
56          output logic [7:0] PPUDO,          // PPU data output
57          input logic [7:0] PPUDI,          // PPU data input
58          output logic [5:0]VGA_STREAM_DATA, // PPU video pipeline out
59          output logic [7:0] PPU_PTR_X,
60          output logic [7:0] PPU_PTR_Y,
61          input logic PPU_SLOW_CLOCK // phase locked ppu slow processing clock
62  );
63  // ========= frame timing parameters =========
64  parameter X_PIXELS = 340;                 // The maximum number of pixels per scanline
65  parameter Y_PIXELS = 262;                // the maximum number of scanlinesh
66  parameter X_BPORCH = 256;                 // start of the x pixel backgporch
67  parameter Y_BPORCH = 240;                 // start of the y pixel backgporch
68  parameter PATTERN_TABLE_0 = 'h0000; // Sprites
69  parameter PATTERN_TABLE_1 = 'h1000; // Backgrounds
70
71  // ============ nametable parameters =========
72  parameter NT_0 = 'h2000;
73  parameter NT_1 = 'h2400;
74  parameter NT_2 = 'h2800;        // NOT NEEDED
75  parameter NT_3 = 'h2C00;        // NOT NEEDED
76  parameter NT_MIRROR = 'h3000;
77
78  // ============ OAM ELEMENT OFFSETs ==============
79  parameter OAM_SPR_YPOS = 0;
80  parameter OAM_SPR_INDX = 1;
81  parameter OAM_SPR_ATTR = 2;
82  parameter OAM_SPR_XPOS = 3;
83
84  // ============ NES REGISTERS ==============
85
86  logic [7:0] PPUCTL = 'b0101_0000;               // 2000 - PPUCTL
87  logic [7:0] PPUMASK;                                                    // 2001
88  logic [7:0] PPUSTATUS;                                                // 2002
89  logic [7:0] OAMADDR;                                                    // 2003
90  logic [7:0] OAMDATA;                                                    // 2004
91  logic [7:0] PPUSCROLL;                                                // 2005
92  logic [7:0] PPUADDR;                                                    // 2006
93  logic [7:0] PPUDATA;                                                    // 2007
94
95
96  // ======== NES register wire assignments =====
97  logic spr_base_rom;
98  logic bkg_base_rom;
99  assign spr_base_rom = PPUCTL[3];
100 assign bkg_base_rom = PPUCTL[4];
101
102 logic [9:0]pixel_x=0;                          // x pixel for fsm
103 logic [9:0]pixel_x_next =0;          // x pixel for fsm
104 logic [7:0]pixel_y=0;                          // y pixel for fsm
105 logic [7:0]pixel_y_next=0;          // y pixel for fsm
106
107 logic [5:0]bkg_cdat=0;                         // output pixel data
```

```
integer i,j,k,l;                                        // general integers for loops

// ===========NT_0 ==========
logic [7:0] NAMETABLE_0[959:0];
logic [7:0] ATTRTABLE_0[63:0];
// ============ OAM ==========
logic [7:0]OAM[255:0];

// ============ sprite render logic ==========
logic [5:0] spr_cdat;
/*
        The sprite render logic consists of two finite state machines that operate in parallel,

        1: spr_scan which fetches data for the next scanline, chr_rom and loads it into the 8 spr_rend_buf

*/

/*
        spr_rend_buf structure map

         |-X|ML| in hex
         bit
         0 - 7  LSB of sprite bitmap (spr_bmp_ls)
         8 - 15 MSB of sprite bitmap (spr_bmp_ms)
         16 - 23 X position of sprite bitmap (spr_bmp_xpos)
         24 - 32 Attribute byte contains extra rendering info
*/
logic [7:0]spr_rend_draw_flags = 0;                 // Draw flags for the next scanline
logic [3:0]spr_rend_pallete_colour [7:0];
logic [7:0]spr_rend_valid;

logic spr_scan_rend_now = 1;
logic [31:0]spr_rend_buf[7:0]; // Sprite draw data for this scanline
logic [31:0]spr_draw_buf[7:0];
logic [7:0]spr_scan_ypos;         // Y position of the current sprite

/*
== CHR rom tile mappings for per slice bitmap fetching
                +-spr_base_rom
                |
                | +-------+- Sprite index
                | |       |
                | |       | +------ msb/lsb select for bitmap
                | |       | |+--+---Sprite slice offset
                | |       | ||  |
        |000R|SSSS|SSSS|BYYY|
        slice_base = '{3'h0,spr_base_rom,OAM[spr_scan_iter << 2],1'b0, y_offset[2:0]};
*/

logic [7:0]spr_tile_index;
logic [2:0]spr_tile_slice;
logic [15:0]spr_tile_slice_ptr;


```

```
// ========== spr_scan fsm =================
parameter SPR_SCAN_SCAN = 0;
parameter SPR_SCAN_HALT = 1;
parameter SPR_REND_FETCH_TILE_LSB=2;
parameter SPR_REND_FETCH_ATTR =3;
parameter SPR_REND_FETCH_DRAW_MSB=4;
parameter SPR_ANIM_TB_0 = 5;
parameter SPR_ANIM_TB_1 = 6;
parameter SPR_ANIM_TB_2 = 7;
parameter SPR_ANIM_TB_3 = 8;

logic [7:0] spr_draw_priority = 0;
logic spr_cdat_fg;
logic [3:0] spr_scan_state = SPR_SCAN_SCAN;
logic [2:0] spr_scan_state_next;
logic [7:0] spr_scan_iter = 0;
logic [2:0] spr_scan_rend_iter = 0;
logic spr_vflip;

// ========== CHR ROM =========
logic [7:0] CHR_ROM_0 ['hFFF:0]; // CHR ROM location
logic [7:0] CHR_ROM_1['hfff:0];        // background CHR

// ========= BKG RENDERING ROM =================
// Consists of name and attribute tables
logic [7:0]NAME_TABLE_0[959:0];
logic [7:0]ATTR_TABLE_0[63:0];

logic [7:0]NAME_TABLE_1[959:0];
logic [7:0]ATTR_TABLE_1[63:0];

// ============= PALLETES ================
// 0-3 is pallete 0
// 4-7 is pallete 1
// 8-B is pallete 2
// C-F is pallete 3

logic [5:0]BKG_PALLETES[15:0];
logic [5:0]SPR_PALLETES[15:0];

// We dont have time to test all programming we are only gonna use preloaded data for this test.
initial begin
        $readmemh("CHR_ROM0.dat", CHR_ROM_0);
        $readmemh("CHR_ROM1.dat", CHR_ROM_1);
        // Auto generated test colours palletes
        BKG_PALLETES[0] = 'h0F; // black
        BKG_PALLETES[1] = 'h15; // white
        BKG_PALLETES[2] = 'h2c; // pink
        BKG_PALLETES[3] = 'h12; // brown

        BKG_PALLETES[4] = 'h0F; // black
        BKG_PALLETES[5] = 'h27; // white
        BKG_PALLETES[6] = 'h02; // turquoise?
```

```
215         BKG_PALLETES[7] = 'h17; // pinkier pink
216
217         BKG_PALLETES[8] = 'h0F; // black
218         BKG_PALLETES[9] = 'h30; // light brown
219         BKG_PALLETES[10] = 'h36; // white
220         BKG_PALLETES[11] = 'h06; // white people
221
222         BKG_PALLETES[12] = 'h0F; // black
223         BKG_PALLETES[13] = 'h30; // brown
224         BKG_PALLETES[14] = 'h2c; // orange
225         BKG_PALLETES[15] = 'h24; // blue
226
227         SPR_PALLETES[0] = 'h0F; // black
228         SPR_PALLETES[1] = 'h02; // grey
229         SPR_PALLETES[2] = 'h36; // blue
230         SPR_PALLETES[3] = 'h16; // red
231
232         SPR_PALLETES[4] = 'h0F; // black
233         SPR_PALLETES[5] = 'h28; // yellow
234         SPR_PALLETES[6] = 'h2A; // green
235         SPR_PALLETES[7] = 'h16; // red
236
237         SPR_PALLETES[8] = 'h0F; // black
238         SPR_PALLETES[9] = 'h2C; // teal
239         SPR_PALLETES[10] = 'h12; // blue
240         SPR_PALLETES[11] = 'h16; // red
241
242         SPR_PALLETES[12] = 'h0F; // black
243         SPR_PALLETES[13] = 'h27; // orange
244         SPR_PALLETES[14] = 'h06; // red
245         SPR_PALLETES[15] = 'h1A; // green
246
247
248         $readmemh("oam_test.dat", OAM);
249         $readmemh("dkimage_NT_0.dat", NAME_TABLE_0);
250         $readmemh("dkimage_AT_0.dat", ATTR_TABLE_0);
251 end
252
253
254 // ========= BKG DRAW FSM =========
255 /*
256         note: the original nintendo PPU doesnt work like this because of limitations at the time. but this
257
258
259         0: FETCHING (happens once per tile, and 32 tiles per scanline)
260         background drawing statemachine
261         on the reset or on pixel_xs where last 3 bits = 0 clock fetch the 2 byte tile slice
262         referenced by the NT at this value and loads it into the
263         output SR, and outputs the 7th pixel
264
265         1: PIPING happens on every pixel that is not a fetch or HALT, shifts out the next pixel to the cdat
```

```
266
267         2: HALT NOT fetching or piping data,
268
269 */
270 parameter FETCHING = 0;
271 parameter PIPING = 1;
272 parameter HALT = 2;
273 parameter BUFF_SLICE_1 = 0;
274 parameter BUFF_SLICE_2 = 1;
275 parameter IDLE = 2;
276 logic [2:0] bkg_draw_state = FETCHING;
277 logic [2:0] bkg_draw_state2 = IDLE;
278
279 logic [4:0] tile_x;                 // tile x coordinate
280 logic [4:0] tile_y;                 // tile y coordinate
281 logic [2:0] tile_col;         // column within a tile
282 logic [2:0] tile_row;         // row within a tile
283 logic [9:0] nt_ptr;                 // name table pointer
284 logic [9:0] nt_ptr_next;// name table pointer for next tile
285 logic [5:0] attr_ptr;         // attribute table pointer
286 logic [15:0] bg_slice;         // two-bite background slice
287 logic [15:0] bg_slice_next;         // two-bite background slice
288 logic [3:0] pallete_ptr='0;// choose colour
289 logic [11:0] chr_ptr_0;          // chr rom pointer
290 logic [11:0] chr_ptr_1;          // chr rom pointer
291 logic [2:0] bkg_offset;
292
293
294
295 //================================================
296 //============ COMBINATIONAL BLOCK===============
297 //================================================
298
299 logic [5:0]draw_cdat;
300 logic spr_cdat_pri;
301 logic bkg_cdat_fg;
302
303 assign PPU_PTR_X = (pixel_x < 256) ? pixel_x : 255;
304 assign PPU_PTR_Y = (pixel_y < 240) ? pixel_y : 239;
305 assign VGA_STREAM_DATA = draw_cdat;
306
307
308 always_comb begin
309 // ------------ PIXELS COUNT INCREMENT -----------
310         if (pixel_x == X_PIXELS-1) begin
311                 pixel_y_next = (pixel_y == Y_PIXELS-1) ? 0 : pixel_y + 1;
312                 pixel_x_next = 0;
313         end
314         else begin
315                         pixel_x_next = pixel_x + 1;
316                         pixel_y_next = pixel_y;
317         end
318 // ------------ background draw state control ----
319         bkg_draw_state = (pixel_y < Y_BPORCH)
```

```
320                 ? (pixel_x[2:0] == 3'b0) ?
321                 FETCHING : PIPING
322         : HALT;
323
324 // ------------ spr_draw_mux ---------
325 // Multi plexer for drawing the combined output of the sprites
326         spr_cdat = 'h0f;
327         draw_cdat = bkg_cdat;
328         spr_cdat_fg = 0;
329         spr_cdat_pri = 0;
330         for ( i = 0; i < 8; i ++) begin
331                 if(spr_rend_valid[i]) begin
332                         spr_cdat = SPR_PALLETES[spr_rend_pallete_colour[i]];
333                         if(spr_rend_pallete_colour[i][1] | spr_rend_pallete_colour[i][0] != 0)
334                         spr_cdat_fg = 1;
335                         if(spr_draw_priority[i])
336                         spr_cdat_pri = 1;
337                 end
338         end
339         tile_x = 0;
340         tile_y = 0;
341         tile_col = 0;
342         tile_row = 0;
343         nt_ptr = 0;
344         attr_ptr = 0;
345 // ------------ Tile coordinates ----------------
346         if (pixel_x < X_BPORCH && pixel_y < Y_BPORCH) begin
347                 tile_x = (pixel_x == 0) ? 0 : (pixel_x-1) >> 3;
348                 tile_y = pixel_y >> 3;
349                 tile_col = pixel_x % 8;
350                 tile_row = pixel_y % 8;
351                 nt_ptr = tile_x + tile_y * 6'd32;
352                 attr_ptr = (tile_x >> 2) + (tile_y >> 2) * 8;
353         end
354
355 // ------------ Output Colour -------------------
356         bkg_cdat = BKG_PALLETES[pallete_ptr];
357         pallete_ptr[1:0] = {bg_slice[7-bkg_offset],bg_slice[15-bkg_offset]};
358
359 // ------------ Attribute decode ----------------
360         if (tile_x % 4 < 2) begin                                       // left side
361                 if (tile_y % 4 < 2) begin                               //top-left
362                 pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][1:0];
363                 end
364                 else begin                                                      // bottom-left
365                 pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][5:4];
366                 end
367         end
368         else begin                                                      // right side
369                 if (tile_y % 4 < 2) begin                               //top-right
370                 pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][3:2];
371                 end
372                 else begin                                                      // bottom-right
373                 pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][7:6];
```

```
374                 end
375         end
376
377 // -------- Next tile pointer -------------------
378         nt_ptr_next = (nt_ptr == 10'd959) ? 0 : nt_ptr + 1;
379 // -------- Sprite and background Colour Data Mux ------
380
381         if(spr_cdat_fg&& (spr_cdat_pri || (pallete_ptr[1:0] == 0))) draw_cdat = spr_cdat;
382
383 end
384 //================================================
385 //================ PER CLK BLOCK  ================
386 //================================================
387 logic [7:0] frame_num = 0;
388 logic [31:0] frame_count = 0;
389 always_ff@(posedge PPU_SLOW_CLOCK)begin
390         pixel_x <= pixel_x_next;
391         pixel_y <= pixel_y_next;
392         frame_count = (frame_count + 1 ) > 1000000 ? 1000000 : frame_count + 1;
393 // NAMETABLE RENDER AND DRAW STATE
394         case(bkg_draw_state)
395                 FETCHING:begin
396                         bg_slice = bg_slice_next;
397                         bkg_offset = '0;
398                 end
399
400                 PIPING: begin
401                         bkg_offset = bkg_offset + 1'b1;
402                 end
403                 HALT: begin
404                         bkg_offset = '0;
405                 end
406         endcase
407 // Sprite spr_scan  Evaluates sprites and loads them into the sprite renderer
408         case(spr_scan_state)
409                 SPR_SCAN_SCAN: begin
410                         spr_scan_rend_now <= 1;
411                         if(spr_scan_rend_iter == 0)
412                         // Scan for sprites that we can draw
413                         spr_rend_draw_flags = 0;
414                         spr_scan_ypos = OAM[spr_scan_iter << 2] + 1;
415
416                         if( (pixel_y < 8||(spr_scan_ypos > (pixel_y - 8))) && (spr_scan_ypos <= (pixel_y ))
417                                 spr_scan_state <= SPR_REND_FETCH_ATTR;
418                         end else begin
419                                 spr_scan_iter = (spr_scan_iter + 1);
420                         end
421
422                         if(spr_scan_iter == 63) begin
423                                 spr_scan_state <= SPR_SCAN_HALT;
424                         end
425                 end
426                 SPR_REND_FETCH_ATTR: begin // grab attributes
427                         spr_rend_buf[spr_scan_rend_iter][31:24] = OAM[(spr_scan_iter << 2) + OAM_SPR_ATTR];
```

```
428                         spr_tile_slice_ptr[12] = spr_base_rom;
429                         spr_vflip = spr_rend_buf[spr_scan_rend_iter][30];
430                         spr_draw_priority[spr_scan_rend_iter] = spr_rend_buf[spr_scan_rend_iter][29];
431                         spr_scan_state <= SPR_REND_FETCH_TILE_LSB;
432                 end
433                 SPR_REND_FETCH_TILE_LSB: begin // grab tile index and tile lsb
434                         spr_tile_index = OAM[(spr_scan_iter << 2) + OAM_SPR_INDX];
435                         spr_tile_slice = pixel_y - spr_scan_ypos;
436                         spr_tile_slice_ptr[15:13] = 0;
437                         spr_tile_slice_ptr[11:4] = spr_tile_index;
438                         spr_tile_slice_ptr[3] = 0;
439                         spr_tile_slice_ptr[2:0] = (spr_vflip) ? 7 - spr_tile_slice : spr_tile_slice ;
440                         spr_rend_buf[spr_scan_rend_iter][7:0] = CHR_ROM_0[spr_tile_slice_ptr];
441                         spr_scan_state <= SPR_REND_FETCH_DRAW_MSB;
442                 end
443                 SPR_REND_FETCH_DRAW_MSB: begin // grab x position and lsb and consolidate and send to sprit
444                         spr_rend_buf[spr_scan_rend_iter][23:16] = OAM[(spr_scan_iter << 2) + OAM_SPR_XPOS];
445                         spr_rend_buf[spr_scan_rend_iter][15:8] = CHR_ROM_0[spr_tile_slice_ptr+8];
446                         spr_rend_draw_flags[spr_scan_rend_iter] = 1;
447                         spr_scan_rend_iter = (spr_scan_rend_iter + 1 < 8) ? spr_scan_rend_iter + 1 : 8;
448                         spr_scan_iter = spr_scan_iter + 1;
449                         if(spr_scan_iter == 63) begin
450                                 spr_scan_state <= SPR_SCAN_HALT;
451                         end else begin
452                                 spr_scan_state <= SPR_SCAN_SCAN;
453                         end
454                 end
455                 SPR_SCAN_HALT: begin
456                         spr_scan_iter = 0;
457                         spr_scan_rend_iter = 0;
458                         spr_scan_rend_now = 0;
459                         if(pixel_x_next == 0) spr_scan_state <= SPR_SCAN_SCAN;
460                         if(frame_count == 1000000) spr_scan_state <= SPR_ANIM_TB_0;
461                 end
462                 SPR_ANIM_TB_0: begin
463                         OAM[(0 << 2) +OAM_SPR_INDX ] = (frame_num << 2) + 0;
464                         spr_scan_state = SPR_ANIM_TB_1;
465                 end
466                 SPR_ANIM_TB_1: begin
467                         OAM[(1 << 2) +OAM_SPR_INDX ] = (frame_num << 2) + 1;
468                         spr_scan_state = SPR_ANIM_TB_2;
469                 end
470                 SPR_ANIM_TB_2: begin
471                         OAM[(2 << 2) +OAM_SPR_INDX ] = (frame_num << 2) + 2;
472                         spr_scan_state = SPR_ANIM_TB_3;
473
474                 end
475                 SPR_ANIM_TB_3: begin
476                         OAM[(3 << 2) +OAM_SPR_INDX ] = (frame_num << 2) + 3;
477                         frame_num = (frame_num + 1 ) % 5;
478                         spr_scan_state = SPR_SCAN_HALT;
479                         frame_count = 0;
480                 end
481         endcase
```

```
482 end
483
484 // ================ SPRITE RENDER MODULES ==============
485
486 spr_rend spr_rend_0( PPU_SLOW_CLOCK, spr_rend_buf[0], pixel_x,
487 spr_rend_draw_flags[0]&spr_scan_rend_now, spr_rend_draw_flags[0],
488 spr_rend_pallete_colour[0], spr_rend_valid[0] );
489 spr_rend spr_rend_1( PPU_SLOW_CLOCK, spr_rend_buf[1], pixel_x,
490 spr_rend_draw_flags[1]&spr_scan_rend_now, spr_rend_draw_flags[1],
491 spr_rend_pallete_colour[1], spr_rend_valid[1] );
492 spr_rend spr_rend_2( PPU_SLOW_CLOCK, spr_rend_buf[2], pixel_x,
493 spr_rend_draw_flags[2]&spr_scan_rend_now, spr_rend_draw_flags[2],
494 spr_rend_pallete_colour[2], spr_rend_valid[2] );
495 spr_rend spr_rend_3( PPU_SLOW_CLOCK, spr_rend_buf[3], pixel_x,
496 spr_rend_draw_flags[3]&spr_scan_rend_now, spr_rend_draw_flags[3],
497 spr_rend_pallete_colour[3], spr_rend_valid[3] );
498 spr_rend spr_rend_4( PPU_SLOW_CLOCK, spr_rend_buf[4], pixel_x,
499 spr_rend_draw_flags[4]&spr_scan_rend_now, spr_rend_draw_flags[4],
500 spr_rend_pallete_colour[4], spr_rend_valid[4] );
501 spr_rend spr_rend_5( PPU_SLOW_CLOCK, spr_rend_buf[5], pixel_x,
502 spr_rend_draw_flags[5]&spr_scan_rend_now, spr_rend_draw_flags[5],
503 spr_rend_pallete_colour[5], spr_rend_valid[5] );
504 spr_rend spr_rend_6( PPU_SLOW_CLOCK, spr_rend_buf[6], pixel_x,
505 spr_rend_draw_flags[6]&spr_scan_rend_now, spr_rend_draw_flags[6],
506 spr_rend_pallete_colour[6], spr_rend_valid[6] );
507 spr_rend spr_rend_7( PPU_SLOW_CLOCK, spr_rend_buf[7], pixel_x,
508 spr_rend_draw_flags[7]&spr_scan_rend_now, spr_rend_draw_flags[7],
509 spr_rend_pallete_colour[7], spr_rend_valid[7] );
510
511 always_ff@(posedge PPU_SLOW_CLOCK) begin
512
513         case(bkg_draw_state2)
514                 BUFF_SLICE_1: begin
515                         chr_ptr_0 = {NAME_TABLE_0[nt_ptr_next],1'b0,tile_row};
516                         bg_slice_next[15:8] = CHR_ROM_1[chr_ptr_0];
517                         bkg_draw_state2 <= BUFF_SLICE_2;
518                 end
519
520                 BUFF_SLICE_2: begin
521                         chr_ptr_1 = {NAME_TABLE_0[nt_ptr_next],1'b1,tile_row};
522                         bg_slice_next[7:0] = CHR_ROM_1[chr_ptr_1];
523                         bkg_draw_state2 <= IDLE;
524                 end
525
526                 IDLE: begin
527                         if(bkg_draw_state == FETCHING)
528                         bkg_draw_state2 <= BUFF_SLICE_1;
529
530                 end
531         endcase
532 end
533 endmodule
```

7 Appendix B - VGA colours

	NES Natives			Corrected VG		
Code	R	G	B	R	G	B
0	75	75	75	6D	6D	6D
1	27	1B	8F	24	24	92
2	00	00	AB	00	00	B6
3	47	00	9F	49	00	92
4	8F	00	77	92	00	6D
5	AB	00	13	B6	00	00
6	A7	00	00	B6	00	00
7	7F	0B	00	6D	00	00
8	43	2F	00	49	24	00
9	00	47	00	00	49	00
0A	00	51	00	00	49	00
0B	00	3F	17	00	49	00
0C	1B	3F	5F	24	49	6D
10	BC	BC	BC	B6	B6	B6
11	00	73	EF	00	6D	FF
12	23	3B	EF	24	49	FF
13	83	00	F3	92	00	FF
14	BF	00	BF	B6	00	B6
15	E7	00	5B	DB	00	6D
16	DB	2B	00	DB	24	00
17	CB	4F	0F	DB	49	00
18	8B	73	00	92	6D	00
19	00	97	00	00	92	00
1A	00	AB	00	00	B6	00
1B	00	93	3B	00	92	49
1C	00	83	8B	00	92	92
20	FF	FF	FF	FF	FF	FF
21	3F	BF	FF	49	B6	FF
22	5F	97	FF	6D	92	FF
23	A7	8B	FD	B6	92	FF
24	F7	7B	FF	FF	6D	FF
25	FF	77	B7	FF	6D	B6
26	FF	77	63	FF	6D	6D
27	FF	9B	3B	FF	92	49
28	F3	BF	3F	FF	B6	49
29	83	D3	13	92	DB	00
2A	4F	DF	4B	49	DB	49
2B	58	F8	98	49	FF	92
2C	00	EB	DB	00	DB	DB
30	FF	FF	FF	FF	FF	FF
31	AB	E7	FF	B6	DB	FF
32	C7	D7	FF	B6	DB	FF
33	D7	CB	FF	DB	DB	FF
34	FF	C7	FF	FF	B6	FF
35	FF	C7	DB	FF	B6	DB
36	FF	BF	B3	FF	B6	B6
37	FF	DB	AB	FF	DB	B6
38	FF	E7	A3	FF	DB	92
39	E3	FF	A3	DB	FF	92
3A	AB	F3	BF	B6	FF	B6
3B	B3	FF	CF	B6	FF	DB
3C	9F	FF	F3	92	FF	FF

VGA RGB VALUES		
0	0	00
1	36	24
2	73	48
3	109	6D
4	146	91
5	182	B6
6	219	DA
7	255	FF

values
[2:0]r
[2:0]g
[2:0]b
Smallest Code Value
0
largest code value
111111
11 1111 6 bits of input

inputs
cCode[5:0]

outputs
[2:0]r
[2:0]g
[2:0]b

Set Colors

Code	Color Verification preconverted	conversion	OUTPUT VALUES R	G	B	Binary Output		
0	#757575	#6D6D6D	3	3	3	011	011	011
1	#271B8F	#242492	1	1	4	001	001	100
2	#0000AB	#0000B6	0	0	5	000	000	101
3	#47009F	#490092	2	0	4	010	000	100
4	#8F0077	#92006D	4	0	3	100	000	011
5	#AB0013	#B60000	5	0	1	101	000	001
6	#A70000	#B60000	5	0	0	101	000	000
7	#7F0B00	#6D0000	3	0	0	011	000	000
8	#432F00	#492400	2	1	0	010	001	000
9	#004700	#004900	0	2	0	000	010	000
0A	#005100	#004900	0	2	0	000	010	000
0B	#003F17	#004900	0	2	1	000	010	001
0C	#1B3F5F	#24496D	1	2	3	001	010	011
10	#BCBCBC	#B6B6B6	5	5	5	101	101	101
11	#0073EF	#006DFF	0	3	7	000	011	111
12	#233BEF	#2449FF	1	2	7	001	010	111
13	#8300F3	#9200FF	4	0	7	100	000	111
14	#BF00BF	#B600B6	5	0	5	101	000	101
15	#E7005B	#DB006D	6	0	3	110	000	011
16	#DB2B00	#DB2400	6	1	0	110	001	000
17	#CB4F0F	#DB4900	6	2	0	110	010	000
18	#8B7300	#926D00	4	3	0	100	011	000
19	#009700	#009200	0	4	0	000	100	000
1A	#00AB00	#00B600	0	5	0	000	101	000
1B	#00933B	#009249	0	4	2	000	100	010
1C	#00838B	#009292	0	4	4	000	100	100
20	#FFFFFF	#FFFFFF	7	7	7	111	111	111
21	#3FBFFF	#49B6FF	2	5	7	010	101	111
22	#5F97FF	#6D92FF	3	4	7	011	100	111
23	#A78BFD	#B692FF	5	4	7	101	100	111
24	#F77BFF	#FF6DFF	7	3	7	111	011	111
25	#FF77B7	#FF6DB6	7	3	5	111	011	101
26	#FF7763	#FF6D6D	7	3	3	111	011	011
27	#FF9B3B	#FF9249	7	4	2	111	100	010
28	#F3BF3F	#FFB649	7	5	2	111	101	010
29	#83D313	#92DB00	4	6	1	100	110	001
2A	#4FDF4B	#49DB49	2	6	2	010	110	010
2B	#58F898	#49FF92	2	7	4	010	111	100
2C	#00EBDB	#00DBDB	0	6	6	000	110	110
30	#FFFFFF	#FFFFFF	7	7	7	111	111	111
31	#ABE7FF	#B6DBFF	5	6	7	101	110	111
32	#C7D7FF	#B6DBFF	5	6	7	101	110	111
33	#D7CBFF	#DBDBFF	6	6	7	110	110	111
34	#FFC7FF	#FFB6FF	7	5	7	111	101	111
35	#FFC7DB	#FFB6DB	7	5	6	111	101	110
36	#FFBFB3	#FFB6B6	7	5	5	111	101	101
37	#FFDBAB	#FFDBB6	7	6	5	111	110	101
38	#FFE7A3	#FFDB92	7	6	4	111	110	100
39	#E3FFA3	#DBFF92	6	7	4	110	111	100
3A	#ABF3BF	#B6FFB6	5	7	5	101	111	101
3B	#B3FFCF	#B6FFDB	5	7	6	101	111	110
3C	#9FFFF3	#92FFFF	4	7	7	100	111	111

case(code)	Green	Blue	HexOut	Code
'h0: R = 3'b011;	'h0: G = 3'b011;	'h0: B = 3'b011;	DB	0
'h1: R = 3'b001;	'h1: G = 3'b001;	'h1: B = 3'b100;	4C	1
'h2: R = 3'b000;	'h2: G = 3'b000;	'h2: B = 3'b101;	5	2
'h3: R = 3'b010;	'h3: G = 3'b000;	'h3: B = 3'b100;	84	3
'h4: R = 3'b100;	'h4: G = 3'b000;	'h4: B = 3'b011;	103	4
'h5: R = 3'b101;	'h5: G = 3'b000;	'h5: B = 3'b001;	141	5
'h6: R = 3'b101;	'h6: G = 3'b000;	'h6: B = 3'b000;	140	6
'h7: R = 3'b011;	'h7: G = 3'b000;	'h7: B = 3'b000;	C0	7
'h8: R = 3'b010;	'h8: G = 3'b001;	'h8: B = 3'b000;	88	8
'h9: R = 3'b000;	'h9: G = 3'b010;	'h9: B = 3'b000;	10	9
'h0A: R = 3'b000;	'h0A: G = 3'b010;	'h0A: B = 3'b000;	10	0A
'h0B: R = 3'b000;	'h0B: G = 3'b010;	'h0B: B = 3'b001;	11	0B
'h0C: R = 3'b001;	'h0C: G = 3'b010;	'h0C: B = 3'b011;	53	0C
'h10: R = 3'b101;	'h10: G = 3'b101;	'h10: B = 3'b101;	16D	10
'h11: R = 3'b000;	'h11: G = 3'b011;	'h11: B = 3'b111;	1F	11
'h12: R = 3'b001;	'h12: G = 3'b010;	'h12: B = 3'b111;	57	12
'h13: R = 3'b100;	'h13: G = 3'b000;	'h13: B = 3'b111;	107	13
'h14: R = 3'b101;	'h14: G = 3'b000;	'h14: B = 3'b101;	145	14
'h15: R = 3'b110;	'h15: G = 3'b000;	'h15: B = 3'b011;	183	15
'h16: R = 3'b110;	'h16: G = 3'b001;	'h16: B = 3'b000;	188	16
'h17: R = 3'b110;	'h17: G = 3'b010;	'h17: B = 3'b000;	190	17
'h18: R = 3'b100;	'h18: G = 3'b011;	'h18: B = 3'b000;	118	18
'h19: R = 3'b000;	'h19: G = 3'b100;	'h19: B = 3'b000;	20	19
'h1A: R = 3'b000;	'h1A: G = 3'b101;	'h1A: B = 3'b000;	28	1A
'h1B: R = 3'b000;	'h1B: G = 3'b100;	'h1B: B = 3'b010;	22	1B
'h1C: R = 3'b000;	'h1C: G = 3'b100;	'h1C: B = 3'b100;	24	1C
'h20: R = 3'b111;	'h20: G = 3'b111;	'h20: B = 3'b111;	1FF	20
'h21: R = 3'b010;	'h21: G = 3'b101;	'h21: B = 3'b111;	AF	21
'h22: R = 3'b011;	'h22: G = 3'b100;	'h22: B = 3'b111;	E7	22
'h23: R = 3'b101;	'h23: G = 3'b100;	'h23: B = 3'b111;	167	23
'h24: R = 3'b111;	'h24: G = 3'b011;	'h24: B = 3'b111;	1DF	24
'h25: R = 3'b111;	'h25: G = 3'b011;	'h25: B = 3'b101;	1DD	25
'h26: R = 3'b111;	'h26: G = 3'b011;	'h26: B = 3'b011;	1DB	26
'h27: R = 3'b111;	'h27: G = 3'b100;	'h27: B = 3'b010;	1E2	27
'h28: R = 3'b111;	'h28: G = 3'b101;	'h28: B = 3'b010;	1EA	28
'h29: R = 3'b100;	'h29: G = 3'b110;	'h29: B = 3'b001;	131	29
'h2A: R = 3'b010;	'h2A: G = 3'b110;	'h2A: B = 3'b010;	B2	2A
'h2B: R = 3'b010;	'h2B: G = 3'b111;	'h2B: B = 3'b100;	BC	2B
'h2C: R = 3'b000;	'h2C: G = 3'b110;	'h2C: B = 3'b110;	36	2C
'h30: R = 3'b111;	'h30: G = 3'b111;	'h30: B = 3'b111;	1FF	30
'h31: R = 3'b101;	'h31: G = 3'b110;	'h31: B = 3'b111;	177	31
'h32: R = 3'b101;	'h32: G = 3'b110;	'h32: B = 3'b111;	177	32
'h33: R = 3'b110;	'h33: G = 3'b110;	'h33: B = 3'b111;	1B7	33
'h34: R = 3'b111;	'h34: G = 3'b101;	'h34: B = 3'b111;	1EF	34
'h35: R = 3'b111;	'h35: G = 3'b101;	'h35: B = 3'b110;	1EE	35
'h36: R = 3'b111;	'h36: G = 3'b101;	'h36: B = 3'b101;	1ED	36
'h37: R = 3'b111;	'h37: G = 3'b110;	'h37: B = 3'b101;	1F5	37
'h38: R = 3'b111;	'h38: G = 3'b110;	'h38: B = 3'b100;	1F4	38
'h39: R = 3'b110;	'h39: G = 3'b111;	'h39: B = 3'b100;	1BC	39
'h3A: R = 3'b101;	'h3A: G = 3'b111;	'h3A: B = 3'b101;	17D	3A
'h3B: R = 3'b101;	'h3B: G = 3'b111;	'h3B: B = 3'b110;	17E	3B
'h3C: R = 3'b100;	'h3C: G = 3'b111;	'h3C: B = 3'b111;	13F	3C
endcase				

The Above spread sheet was used to convert all genuine NES colours into their nearest RGB components. The Following Google Docs code was used to make that all happen.

```
function QUANTIZE(rngs, val) {
  var dv = (rngs[1] - rngs[0])/2;
  var i = 1;
  while((val + dv) >= rngs[i]){
    i++;
  }
  return rngs[i -1];
}

function QUANTIZE_N(rngs, val) {
  var dv = (rngs[1] - rngs[0])/2;
  var i = 1;
  while((val + dv) >= rngs[i]){
    i++;
  }
  return i-1;
}

function setcolor() {
  var cell = SpreadsheetApp.getActiveSheet().getActiveCell();
  var value = cell.getValue(), color = value;
  cell.setBackground(color);
}
```

8 References

[1] P. Diskin. (38200) Nintendo entertainment system documentation. . Accessed 17.04.2017.

[2] S. Larson. (2013) Vga controller (vhdl). https://eewiki.net/pages/viewpage.action?pageId=15925278. Accessed 04/15/2017.

[3] tinyvga.com. (2008) Vga signal 640 x 480 @ 60 hz industry standard timing. http://tinyvga.com/vga-timing/640x480@60Hz. Accessed 4/15/2017.