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.

<table>
<thead>
<tr>
<th>Prepared by:</th>
<th>Prep Date:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter Li</td>
<td>April 18, 2017</td>
</tr>
<tr>
<td>Cameron Morgan</td>
<td>April 18, 2017</td>
</tr>
</tbody>
</table>
# Contents

**List of Figures**

1 **Introduction**
   1.1 Motivations .......................................................... 4
   1.2 Inspirations and other projects ..................................... 4
   1.3 What is the NES ..................................................... 4

2 **A simple look at the nes system and components**
   2.1 Memory and Instructions ............................................ 5

3 **PPU Rendering**
   3.1 PPU memory overview ................................................. 5
   3.2 Bitmap Data storage and encoding .................................. 6
   3.3 Background Rendering and Bitmap decoding ....................... 7
   3.4 Sprites ........................................................................ 9
   3.5 spr and bkg compositing ............................................ 10

4 **VGA video Output**
   4.1 NES colour conversion ................................................ 13

5 **Conclusion**

6 **Appendix A - Code Listings**

7 **Appendix B - VGA colours** ............................................. 43

8 **References** .................................................................. 45
List of Figures

1. Nintendo NES memory management [1] ................................................................. 6
2. How the bits stack together [1] ............................................................................. 7
3. CastleVania’s amazing art despite limitations [1] .................................................. 7
4. Visual representation of the background pattern table for Donkey Kong ............ 8
5. Decoding of Pattern Table Data ........................................................................... 8
6. Decoding of Attribute Table Data ......................................................................... 9
7. Comparison of original game image (Right) and our re-creation (Left) .............. 9
8. VGA timing diagram for h and vsyncs [2] ............................................................... 11
9. VGA timing per scanline timings [3] ..................................................................... 12
10. VGA timing per frame timings [3] ....................................................................... 12
11. VGA 3-bit digital-to-analog circuit ....................................................................... 13
12. NES colours converted to html ......................................................................... 13
13. NES colours converted to html ......................................................................... 14
14. NES colours converted to html ......................................................................... 14
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:

- **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](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/NESS%20emulator%20development%20guide.txt](http://nesdev.com/NESS%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...
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
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
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 tile splits 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.)
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 palette 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, palette 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 versatile than backgrounds in terms of colours available as one of the sprites colours 00 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 NES’s 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 cycles, 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.

**Horizontal Timing**

(h_sync signal)

**Vertical Timing**

(v_sync signal)

---

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

\[ V_{out} = \frac{R_2}{R_2 + R_{vga}} \left( 3.3V \right) = 0.262V \]

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

<table>
<thead>
<tr>
<th>Scanline part</th>
<th>Pixels</th>
<th>Time [\mu s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visible area</td>
<td>640</td>
<td>25.422045680238</td>
</tr>
<tr>
<td>Front porch</td>
<td>16</td>
<td>0.6355114200596</td>
</tr>
<tr>
<td>Sync pulse</td>
<td>96</td>
<td>3.8133068520357</td>
</tr>
<tr>
<td>Back porch</td>
<td>48</td>
<td>1.9066534260179</td>
</tr>
<tr>
<td>Whole line</td>
<td>800</td>
<td>31.777557100298</td>
</tr>
</tbody>
</table>

Figure 9: VGA timing per scanline timings [3]

<table>
<thead>
<tr>
<th>Frame part</th>
<th>Lines</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visible area</td>
<td>480</td>
<td>15.253227408143</td>
</tr>
<tr>
<td>Front porch</td>
<td>10</td>
<td>0.31777557100298</td>
</tr>
<tr>
<td>Sync pulse</td>
<td>2</td>
<td>0.06355114200596</td>
</tr>
<tr>
<td>Back porch</td>
<td>33</td>
<td>1.0486593843098</td>
</tr>
<tr>
<td>Whole frame</td>
<td>525</td>
<td>16.683217477656</td>
</tr>
</tbody>
</table>

Figure 10: VGA timing per frame timings [3]
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>preconverted</th>
<th>conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#757575</td>
<td>#D6D6D6</td>
</tr>
<tr>
<td>1</td>
<td>#271B8F</td>
<td>#242492</td>
</tr>
<tr>
<td>2</td>
<td>#0000AB</td>
<td>#0000B6</td>
</tr>
<tr>
<td>3</td>
<td>#7F000F</td>
<td>#990000</td>
</tr>
<tr>
<td>4</td>
<td>#000000</td>
<td>#000000</td>
</tr>
<tr>
<td>5</td>
<td>#000000</td>
<td>#000000</td>
</tr>
<tr>
<td>6</td>
<td>#000000</td>
<td>#000000</td>
</tr>
<tr>
<td>7</td>
<td>#000000</td>
<td>#000000</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Code</th>
<th>preconverted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#757575</td>
</tr>
<tr>
<td>1</td>
<td>#271B8F</td>
</tr>
<tr>
<td>2</td>
<td>#0000AB</td>
</tr>
<tr>
<td>3</td>
<td>#4709F</td>
</tr>
<tr>
<td>4</td>
<td>#8F0077</td>
</tr>
<tr>
<td>5</td>
<td>#AB0013</td>
</tr>
<tr>
<td>6</td>
<td>#A70000</td>
</tr>
<tr>
<td>7</td>
<td>#7F0B00</td>
</tr>
<tr>
<td>8</td>
<td>#432F00</td>
</tr>
<tr>
<td>9</td>
<td>#004700</td>
</tr>
</tbody>
</table>

Figure 13: NES colours converted to html

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

<table>
<thead>
<tr>
<th>conversion</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>Binary Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>#6D6D6D</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>#242492</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>001</td>
</tr>
<tr>
<td>#0000B6</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>000</td>
</tr>
<tr>
<td>#490092</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>010</td>
</tr>
<tr>
<td>#92006D</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>#B60000</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>#B60000</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>#6D0000</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>011</td>
</tr>
</tbody>
</table>

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.
Appendix A - Code Listings

```verbatim
// 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] pallelete_colour // 4 bit output to pallelete
    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] pallelete_colour, // 4 bit output to pallelete
    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] pallelete = 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 pallelete_colour = {pallelete, bmp_output};

always_ff @(posedge rend_now)begin
    pallelete = {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];
endmodule
```
always_ff
@ (negedge clk) begin
if((pixel_x >= xpos) && (pixel_x < xpos + 8) && draw) begin
valid = 1;
vslice = pixel_x - xpos;
bmp_output = {bmp_data[vslice+8], bmp_data[vslice]};
end
else valid = 0;
end
module nes_video_dc_fifo // This thing does NOT check for errors
  input logic [31:0] din,
  input logic clk_write,
  input logic read,
  input logic write,
  input logic clk_read,
  output logic [5:0] dout,
  input logic reset,
  output logic done
);

logic [5:0] vid_fifo[512:0]; // size be a bit overkill
logic ptr_read = 0;
logic ptr_write = 0;

assign done = (ptr_write == ptr_read);

// if write is high ptr will increment and data will be pushed in
always_ff@ (posedge clk_write) begin
  if(reset) begin
    ptr_write = 0;
  end else begin
    if (write)
      vid_fifo[ptr_write] = {din[5:0]};
    ptr_write = ptr_write + 1;
  end
end

// if read is high, ptr will increment and data will be pushed out
always_ff@ (posedge clk_read) begin
  if(reset) begin
    ptr_read = 0;
  end else begin
    if (read && (!done))
      dout = vid_fifo[ptr_read];
    if (!done)
      ptr_read = ptr_read + 1;
  end
end

endmodule

// VGA ROM top level test.

// Tests the following:
// VGA output operation - working but glitchy
// NES colour decoding - passed
// Right now we need to solve the "square problem"
// some anomalies are observed on the right side of the screen

// This thing also tests uart_port TX and RX loopback

// (PPU REPLACED BY TEST_IMG_DUMMY_ROM) ->
module vga_rom_test_top(
    input logic uart_port_DI, // gpio 31
    output logic uart_port_DO, // gpio 33
    input logic CLOCK_50, rst,
    output logic [2:0]RED,
    output logic [2:0]GREEN,
    output logic [2:0]BLUE,
    output logic HSYNC, VSYNC);

// ====== RX signals ======
logic [15:0]read_ptr; // buffer read pointer
logic rx_clear;
logic read_valid;
logic [7:0]uart_DO;

// ====== tx signals ======
logic [15:0]send_ptr;
logic tx_clear;
logic [7:0]tx_DI;
logic send_done;

uart_port uart_0(.clk(ppu_clk),..*);

logic nios_clk;
logic pix_clk;
logic ppu_clk;
logic ppu_slow_clk;
logic A203_clk;

logic [5:0]test_read_col[240];

logic [7:0]fb_ptr_x;
logic [7:0]fb_ptr_y;

logic [7:0]ppu_ptr_x;
logic [7:0]ppu_ptr_y;
logic [8:0]rgb;
logic [5:0]testColours;
// PIN outputs
logic vsync;
logic hsync;
logic [8:0]rgb_OUT;

assign RED = rgb_OUT[8:6];
assign GREEN = rgb_OUT[5:3];
assign BLUE = rgb_OUT[2:0];
assign VSYNC = vsync;
assign HSYNC = hsync;

// Test variables
integer i, j, k;
logic [7:0]grid_center; // rolls over at 128
logic [7:0]h_line = 120; // rolls over at 128
logic [7:0]y_line = 0; // rolls over at 128
logic [7:0]h_line_values[3:0];
logic [7:0]y_line_values[3:0];
integer n_line = 0;
integer progState;

assign h_line = grid_center;
assign y_line = grid_center;
initial begin
    h_line_values[0] = 0;
    h_line_values[1] = 255 - 10;
    h_line_values[2] = 255 - 10;
    h_line_values[3] = 0;
    y_line_values[0] = 0;
    y_line_values[1] = 000;
    y_line_values[2] = 239 - 10;
    y_line_values[3] = 239 - 10;
end
always_ff @(posedge ppu_clk) begin
    if(progState < 256) begin
        // state 0 = reset
        i = 0;
        progState = progState + 1;
    end else begin
        if(i < (240*256 + 1)) begin
            if(ppu_ptr_x == 255)
                ppu_ptr_y = ppu_ptr_y + 1;
            ppu_ptr_x = ppu_ptr_x+1;

            if(ppu_ptr_x < y_line) begin
                if(ppu_ptr_y < h_line)
                    testColours = [h27; // Orange
                                  else
                        testColours = [h14; // magenta

                else begin
                    if(ppu_ptr_y < h_line)
                        testColours = [h01; // blue
                                  else
                        // Other cases...
end
end
testColours = h2b; // green

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

// 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
```verilog
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
asynch_out

settings for com port
115200 baud
1 start bit
no parity
1 stop bit (or more, it doesn't matter on RX side, tx will output 1)
*/
8 bit data

--- How to use ---

- instantiate uart_port()
simply connect your UART signal to uart-port-DI

tx data can be sent by writing to the buffer then incrementing the send_ptr,
available data will be present in the 64kB buffer, this is dereferenced via the read_ptr
to prevent buffer from overflowing hold CLEAR signal high
to read after X number of bits are available set read_ptr
to X and wait for read_valid

*/

module uart_port( // instantiates the entire port
    input logic clk,
    // ====== RX signals ======
    input logic [15:0] read_ptr, // buffer read pointer
    input logic rx_clear,
    output logic read_valid,
    output logic [ 7:0] uart_DO,
    // ====== tx signals ======
    input logic [15:0] send_ptr,
    input logic tx_clear,
    input logic [ 7:0] tx_DI,
    output logic send_done,
    // ===== Phyiscal output pins ======
    input logic uart_port_DI,
    output logic uart_port_DO
);

logic uart_valid;
logic [7:0]uart_rx_DO;
uart_tx uart_transmitter(
    .send_ptr, .uart_port_DO,
    .clear(tx_clear), .tx_DI,
    .send_done, .clk
);

uart_buf uart_buffer(
    .read_ptr, .uart_DI(uart_rx_DO),
    .read_clk(clk), .uart_valid,
    .clear(rx_clear), .uart_DO(uart_port_DO),
    .read_valid
);

uart_rx uart_reciever(
    .uart_DI(uart_port_DI),
.clk,
.uart_valid,
.uart_DO(uart_rx.DO)
);
endmodule

module uart_buf(/\ 64kB UART read buffer
  input logic [15:0]read_ptr,
  input logic [7:0]uart_DI,
  input logic uart_valid, read_clk,
  input logic clear, // if purge is pulled high, the rx_ptr will go to 0
  output logic[7:0]uart_DO,
  output logic read_valid
);
logic [15:0]rx_ptr = 0;
logic [15:0]rx_ptr_next = 0;
logic [7:0]rx_buf[1600:0];
// On each read_clk data is read out to the controlling module
always_ff(posedge read_clk) begin
  uart_DO = rx_buf[read_ptr];
  if(clear) begin
    rx_ptr = 0;
  end else begin
    rx_ptr = rx_ptr_next;
  end
end
assign read_valid = (rx_ptr >= read_ptr);
// data is valid so long as the rx_ptr also if the buffer is full then all data locations are valid
// is greater than the requested data location
// Data is read into the current ptr location and the pointer is incremented
always_ff(posedge uart_valid)begin
  rx_buf[rx_ptr] = uart_DI;
  rx_ptr_next = rx_ptr + 1;
end
endmodule

module uart_rx( /\ uart in and parallel 8 bit out
  input logic uart_DI,
  input logic clk, // operates at will borrow a 21.477 mhz clk borrowed from the ppu_clk
  output logic uart_valid,
  output logic [7:0]uart_DO
);
parameter NCLKS_PER_BIT = 186; // 21 477 000/(115200) ~= 186.4 Cycles of oversampling
// Depending on FTDI settings this could be different
parameter SPACE = 1;
parameter MARK = 0;
parameter IDLE = SPACE; // IDLE is always space
parameter START = !IDLE; // start is always the opposite of space
parameter STOP = !START; // stop bit is always the opposite of start
// RX machine states
parameter WAITING = 0;
parameter READING_DI = 1;
parameter STOPPING = 2;
parameter START_BIT = 3;
logic [2:0]bit_ptr = 0;
logic [1:0]state = WAITING;
logic [15:0]count = 0;

// RX fsm
always_ff (posedge clk) begin
  case(state)
    WAITING: begin
      uart_valid <= 0;
      if(uart_DI == START) begin
        state <= START_BIT;
        count <= 0;
      end
    end
    START_BIT: begin
      if(count == NCLKS_PER_BIT * 276) begin
        count <= 0;
        bit_ptr <= bit_ptr + 1;
        uart_DO[bit_ptr] <= uart_DI ^ SPACE;
      end else begin
        count <= count + 1;
      end
    end
    READING_DI: begin
      if(count == NCLKS_PER_BIT) begin
        count <= 0;
        bit_ptr <= bit_ptr + 1;
        uart_DO[bit_ptr] <= uart_DI ^ SPACE;
        if(bit_ptr == 7) state <= STOPPING;
      end else begin
        count <= count + 1;
      end
    end
    STOPPING: begin
      bit_ptr <= 0; // Should already be at zero but this will ensure
      if(count > NCLKS_PER_BIT/2)
        uart_valid <= 1;
      if(count == NCLKS_PER_BIT)begin
        count <=0;
        state <= WAITING;
      end else begin
        count <= count + 1;
      end
  endcase
module uart_tx(
    input logic clk, clear, // Same clock as RX
    input logic [7:0]tx_DI,
    input logic [15:0]send_ptr,
    output logic uart_port_DO,
    output logic send_done
);

parameter NCLKS_PER_BIT = 186; // 21,477,000/(115200) ~=
parameter NCLKS_PER_BIT_1_5 = 276;
// 186.4 Cycles of oversampling

// Depending on FTDI settings this could be different
parameter SPACE = 1;
parameter MARK = 0;
parameter IDLE = SPACE; // IDLE is always space
parameter START = !IDLE; // start is always the opposite of
// space
parameter STOP = !START; // stop bit is always the opposite of start

parameter WAITING = 0;
parameter SENDING_DO = 1;
parameter STOPPING = 2;
parameter START_BIT = 3;

logic [7:0]tx_buf[7:0]; // 8 byte Output buffer
logic [7:0]tx_ptr = 0; // data control pointer
logic [2:0]tx_bit_ptr = 0; // pointer for bitwise send
logic [15:0]count = 0; // Timing Counter for
logic [1:0]state = WAITING; // state for the FSM

// Per byte tx logic (fsm)
always_ff @ (posedge clk) begin
    tx_buf[send_ptr] = tx_DI; //data in buffer isn't considered valid unless the send pointer has increased
    send_done = (tx_ptr >= send_ptr); //data in buffer isn't considered valid unless the send pointer has increased
    if (clear) begin
        tx_ptr = 0;
    end
    case(state)
        WAITING: begin
            uart_port_DO <= SPACE;
            // The user will increment
            // send_ptr as soon as there
            // is data
            if (send_ptr > tx_ptr) begin
                count <= 0;
                state <= START_BIT;
            end
        end
    endcase
end
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
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
logic [15:0] send_ptr = 0;
logic tx_clear = 1;
logic [7:0] tx_DI = 0;
logic send_done;
logic uart_port_DO;

// OUTPUTS
logic [7:0] uart_DO;
logic read_valid;

logic [7:0] ascii_char = 65; // ascii char 'a'
// test vars

integer i, j, k, l;

uart_port dut(.*);
initial begin
    clk = 0;
    rx_clear = 0;

    // while(!read_valid) begin
    #8.68us; // Send Start Bit
    uart_port_DI = 0;
    for (i = 0; i < 8; i++) begin
        #8.68us;
        uart_port_DI = !ascii_char[i];
    end
    #8.68us; // Send Stop Bit
    uart_port_DI = 1;
    ascii_char = ascii_char + 1;

    end
    /*
    // END OF UARTx test
    ascii_char = 'h41'; // reset back to 'A'
    tx_clear = 0;
    tx_DI = ascii_char;
    ascii_char = ascii_char + 1;
    #46.2ns;
    send_ptr = send_ptr + 1;
    tx_DI = ascii_char;
    #46.2ns;
    send_ptr = send_ptr + 1;
    tx_clear = 1;
    send_ptr = 1; // blast last transmission.
    #86.8us
    // END OF UARTx test
    $stop;
    */
    end

always begin
    #4.34us;
    uart_clk = ~uart_clk;
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                   // horizontal 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;
```vhdl
always_comb begin
    rgb = (pixel_x >= NES_W && pixel_x < R_BLANK && pixel_y < VF_PORCH) ? rgb_buf : 0;
    if(pixel_x >= R_BLANK) rgb = 0;
end

always_ff @(posedge pix_clk) begin
    // RGB control
    if (pixel_x >= L_BLANK && pixel_x < NES_W) rgb <= '0;
    if (pixel_x >= NES_W && pixel_x < R_BLANK && pixel_y < VF_PORCH) begin
        // RGB gets NES info // Get the information from the framebuffer module
        rgb = rgb_buf;
    end
    if (pixel_x >= R_BLANK) rgb <= '0;

    // HSYNC Control
    if (pixel_x >= HSYNC_START && pixel_x < HB_PORCH) hsync <= 0;
    else hsync <= 1;

    // VSYNC Control
    if (pixel_y >= VSYNC_START && pixel_y < VB_PORCH) vsync <= 0;
    else vsync <= 1;

    // move to next pixel
    if (pixel_x == 'd399) begin
        // reset x value
        pixel_x <= '0;
        // increment or reset y value
        if (pixel_y == 'd524)
            pixel_y <= '0;
        else
            pixel_y <= pixel_y + 1'b1;
    end
    else
        pixel_x <= pixel_x + 1'b1;
end

always_comb begin
    if (pixel_x >= NES_W && pixel_x < R_BLANK) // before or after visible area
        pix_ptr_x = pixel_x - NES_W;
    else
        pix_ptr_x = 0; // set pointer for next pixel to be rendered

    // lines are doubled to fill the screen
    if (pixel_y < VF_PORCH)
        pix_ptr_y = pixel_y >> 1; // right-shift will duplicate lines
```
else
    pix_ptr_y = 0;
end
endmodule

DB 4C 5 84 103 141 140 CO 88 10 10 11 53 00 00 00 16D 1F 57 107 145 183 188 190 118 20 28 22 24 00 00 00 1FF AF E7 167 1DF 1DD 1DB 1E2 1EA 131 B2 BC 36 00 00 00 1FF
*/
 Frame buffer relationships:
 +---------------+ c_codes +---------------+
 | | (eg. 18 | |
 | | +--------> | vga | |
 | ppu | gives a | Frame |
 | | brownish | Buffer |
 | | colour | |
 | | +--------+
 +---------------+ +---------------+
 |rgb values |
 |RRRGGGBBBB |
 |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
input logic ppu_ctl_clk,
input logic [7:0] ppu_ptr_x,
input logic [7:0] ppu_ptr_y,
// TODO The ppu may not write each byte individually
// Investigate this
input logic [5:0] ppu_DI,
input logic CS,
// Output to vga_out Module
input logic pix_clk,
input logic [7:0] pix_ptr_x,
input logic [7:0] pix_ptr_y,
output logic [8:0] rgb // format of RRRGGBBB r is always gonna be msb
);
// logic [5:0] pixel_code [239:0] [255:0]; // Frame buffer RAM

logic [5:0] pix;
logic [2:0] r;
logic [2:0] g;
logic [2:0] b;
logic [8:0] coloursDecode[63:0];
logic [8:0] dec;

logic [7:0] pix_ptr_y_clamp;
logic [7:0] ppu_ptr_y_clamp;

assign pix_ptr_y_clamp = pix_ptr_y > 239 ? 239 : pix_ptr_y;
assign ppu_ptr_y_clamp = ppu_ptr_y > 239 ? 239 : ppu_ptr_y;

sc_dpram (
    pix_clk,
    ppu_DI,
    {pix_ptr_y_clamp, pix_ptr_x}, // read address
    {ppu_ptr_y_clamp, ppu_ptr_x}, // write address
    [h1, pix);

initial begin
  $readmemh("vga_colours_rgb.txt",coloursDecode);
end

/*always_ff@ (posedge pix_clk) begin
    pix = pixel_code [ppu_ptr_y_clamp][ppu_ptr_x];
end*/

// PPU access (Write only)
always_ff@ (posedge pix_clk) begin
    pixel_code [ppu_ptr_y_clamp][ppu_ptr_x] = ppu_ptr_x > 127?'h27:'h15;
end

assign dec = coloursDecode[pix];
// vga out access (Read only)
assign rgb = {dec[8:6], dec[5:3], dec[2:0]};
endmodule
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-£2007
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
output logic [13:0] APPU, // Address and data pins
output logic [7:0] PPUUDO, // PPU data output
input logic [7:0] PPUDI, // PPU data input
output logic [5:0] VGA_STREAM_DATA, // PPU video pipeline out
output logic [7:0] PPU_PTR_X,
output logic [7:0] PPU_PTR_Y,
input logic PPU_SLOW_CLOCK // phase locked ppu slow processing clock
);

// ========= frame timing parameters =========
parameter X_PIXELS = 340; // The maximum number of pixels per scanline
parameter Y_PIXELS = 262; // the maximum number of scanlines
parameter X_BPORCH = 256; // start of the x pixel backgporch
parameter Y_BPORCH = 240; // start of the y pixel backgporch
parameter PATTERN_TABLE_0 = 'h0000; // Sprites
parameter PATTERN_TABLE_1 = 'h1000; // Backgrounds

// ============= nametable parameters ==============
parameter NT_0 = 'h2000;
parameter NT_1 = 'h2400;
parameter NT_2 = 'h2800; // NOT NEEDED
parameter NT_3 = 'h2C00; // NOT NEEDED
parameter NT_MIRROR = 'h3000;

// =========== OAM ELEMENT OFFSETs ============
parameter OAM_SPR_YPOS = 0;
parameter OAM_SPR_INDX = 1;
parameter OAM_SPR_ATTR = 2;
parameter OAM_SPR_XPOS = 3;

// =========== NES REGISTERS ============
logic [7:0] PPUCTL = 'b0101_0000; // 2000 - PPUCTL
logic [7:0] PPUMASK; // 2001
logic [7:0] PPUSTATUS; // 2002
logic [7:0] OAMADDR; // 2003
logic [7:0] OAMDATA; // 2004
logic [7:0] PPUSCROLL; // 2005
logic [7:0] PPUADDR; // 2006
logic [7:0] PPUDATA; // 2007

// ========= NES register wire assignments =====
logic spr_base_rom;
logic bkg_base_rom;
assign spr_base_rom = PPUCTL[3];
assign bkg_base_rom = PPUCTL[4];
logic [9:0] pixel_x=0; // x pixel for fsm
logic [9:0] pixel_x_next =0; // x pixel for fsm
logic [7:0] pixel_y=0; // y pixel for fsm
logic [7:0] pixel_y_next=0; // y pixel for fsm
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_pallette_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
   sprite_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 [0x0]hFFF:0; // CHR ROM location
logic [7:0] CHR_ROM_1 [0x0]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 pallele 1
// 8-B is pallele 2
// C-F is pallele 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?
// ========= BKG DRAW FSM =========
/*
note: the original nintendo PPU doesn't work like this because of limitations at the time. but this

0: FETCHING (happens once per tile, and 32 tiles per scanline)
background drawing state machine
on the reset or on pixel xs where last 3 bits = 0 clock fetch the 2 byte tile slice
referenced by the NT at this value and loads it into the
output SR, and outputs the 7th pixel

1: PIPING happens on every pixel that is not a fetch or HALT, shifts out the next pixel to the cdat
2: HALT NOT fetching or piping data,

parameter FETCHING = 0;
parameter PIPING = 1;
parameter HALT = 2;
parameter BUFF_SLICE_1 = 0;
parameter BUFF_SLICE_2 = 1;
parameter IDLE = 2;
logic [2:0] bkg_draw_state = FETCHING;
logic [2:0] bkg_draw_state2 = IDLE;
logic [4:0] tile_x; // tile x coordinate
logic [4:0] tile_y; // tile y coordinate
logic [2:0] tile_col; // column within a tile
logic [2:0] tile_row; // row within a tile
logic [9:0] nt_ptr; // name table pointer
logic [9:0] nt_ptr_next; // name table pointer for next tile
logic [5:0] attr_ptr; // attribute table pointer
logic [15:0] bg_slice; // two-bite background slice
logic [15:0] bg_slice_next; // two-bite background slice
logic [3:0] pallete_ptr[0]; // choose colour
logic [11:0] chr_ptr_0; // chr rom pointer
logic [11:0] chr_ptr_1; // chr rom pointer
logic [2:0] bkg_offset;

//===============================================
//============ COMBINATIONAL BLOCK===============
//===============================================

logic [5:0] draw_cdat;
logic spr_cdat_pri;
logic bkg_cdat_fg;

assign PPU_PTR_X = (pixel_x < 256) ? pixel_x : 255;
assign PPU_PTR_Y = (pixel_y < 240) ? pixel_y : 239;
assign VGA_STREAM_DATA = draw_cdat;

always_comb begin
    // ----------- PIXELS COUNT INCREMENT -----------
    if (pixel_x == X_PIXELS-1) begin
        pixel_y_next = (pixel_y == Y_PIXELS-1) ? 0 : pixel_y + 1;
    end
    else begin
        pixel_x_next = pixel_x + 1;
        pixel_y_next = pixel_y;
    end
    // ----------- background draw state control ----
    bkg_draw_state = (pixel_y < Y_BPORCH)
? (pixel_x[2:0] == 3'h0) ?
FETCHING : PIPING
: HALT;

// ------------ spr_draw_mux ----------
// Multi plexer for drawing the combined output of the sprites
spr_cdat = h'0f;
draw_cdat = bkg_cdat;
spr_cdat_fg = 0;
spr_cdat_pri = 0;
for ( i = 0; i < 8; i ++ ) begin
  if(spr_rend_valid[i]) begin
    spr_cdat = SPR_PALLETES[spr_rend_pallete_colour[i]];
    if(spr_rend_pallete_colour[i][1] | spr_rend_pallete_colour[i][0] != 0)
      spr_cdat_fg = 1;
    if(spr_draw_priority[i])
      spr_cdat_pri = 1;
  end
end
tile_x = 0;
tile_y = 0;
tile_col = 0;
tile_row = 0;
nt_ptr = 0;
attr_ptr = 0;

// ------------ Tile coordinates ---------------
if (pixel_x < X_BPORCH && pixel_y < Y_BPORCH) begin
  tile_x = (pixel_x == 0) ? 0 : (pixel_x-1) >> 3;
tile_y = pixel_y >> 3;
tile_col = pixel_x % 8;
tile_row = pixel_y % 8;
  nt_ptr = tile_x + tile_y * 6'h32;
  attr_ptr = (tile_x >> 2) + (tile_y >> 2) * 8;
end

// ------------ Output Colour ---------------
bkg_cdat = BKG_PALLETES[pallete_ptr];
pallete_ptr[1:0] = {bg_slice[7-bkg_offset],bg_slice[15-bkg_offset]};

// ------------ Attribute decode ---------------
if (tile_x % 4 < 2) begin
  if (tile_y % 4 < 2) begin
    pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][1:0];
  end
  else begin
    pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][5:4];
  end
else begin
  pallete_ptr[3:2] = ATTR_TABLE_0[attr_ptr][7:6];
end
// -------- Next tile pointer -------------------
nt_ptr_next = (nt_ptr == 10'bd959) ? 0 : nt_ptr + 1;

// -------- Sprite and background Colour Data Mux -------
if(spr_cdat_fg && (spr_cdat_pri || (pallete_ptr[1:0] == 0))) draw_cdat = spr_cdat;
end

logic [7:0] frame_num = 0;
logic [31:0] frame_count = 0;
always_ff @ (posedge PPU_SLOW_CLOCK) begin
    pixel_x <= pixel_x_next;
    pixel_y <= pixel_y_next;
    frame_count = (frame_count + 1) > 1000000 ? 1000000 : frame_count + 1;
end

// NAMETABLE RENDER AND DRAW STATE
case (bkg_draw_state)
    FETCHING: begin
        bg_slice = bg_slice_next;
        bkg_offset = ['0';
    end

    PIPING: begin
        bkg_offset = bkg_offset + 1'b1;
    end

    HALT: begin
        bkg_offset = ['0';
    end
endcase

// Sprite spr_scan Evaluates sprites and loads them into the sprite renderer
case (spr_scan_state)
    SPR_SCAN_SCAN: begin
        spr_scan_rend_now <= 1;
        if (spr_scan_rend_iter == 0)

        // Scan for sprites that we can draw
        spr_rend_draw_flags = 0;
        spr_scan_ypos = OAM[spr_scan_iter << 2] + 1;

        if (pixel_y < 8 || (spr_scan_ypos > (pixel_y - 8))) && (spr_scan_ypos <= (pixel_y + 8))
            spr_scan_state <= SPR_REND_FETCH_ATTR;
       else begin
            spr_scan_iter = (spr_scan_iter + 1);
        end

        if (spr_scan_iter == 63) begin
            spr_scan_state <= SPR_SCAN_HALT;
        end

    SPR_REND_FETCH_ATTR: begin // grab attributes
        spr_rend_buf[spr_scan_rend_iter][31:24] = OAM[(spr_scan_iter << 2) + OAM_SPR_ATTR];
    end
endcase
spr_tile_slice_ptr[12] = spr_base_rom;
spr_vflip = spr_rend_buf[spr_scan_render_iter][30];
spr_draw_priority[spr_scan_render_iter] = spr_rend_buf[spr_scan_render_iter][29];
spr_scan_state <= SPR_RENDER_FETCH_TILE_LSB;
end

SPR_RENDER_FETCH_TILE_LSB: begin // grab tile index and tile lsb
  spr_tile_index = OAM[(spr_scan_iter << 2) + OAM_SPR_INDX];
  spr_tile_slice = pixel_y - spr_scan_ypos;
  spr_tile_slice_ptr[15:13] = 0;
  spr_tile_slice_ptr[11:4] = spr_tile_index;
  spr_tile_slice_ptr[3] = 0;
  spr_tile_slice_ptr[2:0] = (spr_vflip) ? 7 - spr_tile_slice : spr_tile_slice ;
  spr_rend_buf[spr_scan_render_iter][7:0] = CHR_ROM_0[spr_tile_slice_ptr];
  spr_scan_state <= SPR_RENDER_FETCH_DRAW_MSB;
end

SPR_RENDER_FETCH_DRAW_MSB: begin // grab x position and lsb and consolidate and send to sprite
  spr_rend_buf[spr_scan_render_iter][23:16] = OAM[(spr_scan_iter << 2) + OAM_SPR_XPOS];
  spr_rend_buf[spr_scan_render_iter][15:8] = CHR_ROM_0[spr_tile_slice_ptr+8];
  spr_rend_draw_flags[spr_scan_render_iter] = 1;
  spr_scan_render_iter = (spr_scan_render_iter + 1 < 8) ? spr_scan_render_iter + 1 : 8;
  spr_scan_iter = spr_scan_iter + 1;
  if(spr_scan_iter == 63) begin
    spr_scan_state <= SPR_SCAN_HALT;
  end else begin
    spr_scan_state <= SPR_RENDER_SCAN;
  end
end

SPR_SCAN_HALT: begin
  spr_scan_iter = 0;
  spr_scan_render_iter = 0;
  spr_scan_render_now = 0;
  if(pixel_x_next == 0) spr_scan_state <= SPR_RENDER_SCAN;
  if(frame_count == 1000000) spr_scan_state <= SPR_ANIM_TB_0;
end

SPR_ANIM_TB_0: begin
  OAM[(0 << 2) + OAM_SPR_INDX ] = (frame_num << 2) + 0;
  spr_scan_state = SPR_ANIM_TB_1;
end

SPR_ANIM_TB_1: begin
  OAM[(1 << 2) + OAM_SPR_INDX ] = (frame_num << 2) + 1;
  spr_scan_state = SPR_ANIM_TB_2;
end

SPR_ANIM_TB_2: begin
  OAM[(2 << 2) + OAM_SPR_INDX ] = (frame_num << 2) + 2;
  spr_scan_state = SPR_ANIM_TB_3;
end

SPR_ANIM_TB_3: begin
  OAM[(3 << 2) + OAM_SPR_INDX ] = (frame_num << 2) + 3;
  frame_num = (frame_num + 1 ) % 5;
  spr_scan_state = SPR_RENDER_SCAN;
  frame_count = 0;
end
endcase
// ================ SPRITE RENDER MODULES ==============

spr_rend spr_rend_0( PPU_SLOW_CLOCK, spr_rend_buf[0], pixel_x, spr_rend_draw_flags[0]&spr_scan_rend_now, spr_rend_draw_flags[0], spr_rend_pallete_colour[0], spr_rend_valid[0] );
spr_rend spr_rend_1( PPU_SLOW_CLOCK, spr_rend_buf[1], pixel_x, spr_rend_draw_flags[1]&spr_scan_rend_now, spr_rend_draw_flags[1], spr_rend_pallete_colour[1], spr_rend_valid[1] );
spr_rend spr_rend_2( PPU_SLOW_CLOCK, spr_rend_buf[2], pixel_x, spr_rend_draw_flags[2]&spr_scan_rend_now, spr_rend_draw_flags[2], spr_rend_pallete_colour[2], spr_rend_valid[2] );
spr_rend spr_rend_3( PPU_SLOW_CLOCK, spr_rend_buf[3], pixel_x, spr_rend_draw_flags[3]&spr_scan_rend_now, spr_rend_draw_flags[3], spr_rend_pallete_colour[3], spr_rend_valid[3] );
spr_rend spr_rend_4( PPU_SLOW_CLOCK, spr_rend_buf[4], pixel_x, spr_rend_draw_flags[4]&spr_scan_rend_now, spr_rend_draw_flags[4], spr_rend_pallete_colour[4], spr_rend_valid[4] );
spr_rend spr_rend_5( PPU_SLOW_CLOCK, spr_rend_buf[5], pixel_x, spr_rend_draw_flags[5]&spr_scan_rend_now, spr_rend_draw_flags[5], spr_rend_pallete_colour[5], spr_rend_valid[5] );
spr_rend spr_rend_6( PPU_SLOW_CLOCK, spr_rend_buf[6], pixel_x, spr_rend_draw_flags[6]&spr_scan_rend_now, spr_rend_draw_flags[6], spr_rend_pallete_colour[6], spr_rend_valid[6] );
spr_rend spr_rend_7( PPU_SLOW_CLOCK, spr_rend_buf[7], pixel_x, spr_rend_draw_flags[7]&spr_scan_rend_now, spr_rend_draw_flags[7], spr_rend_pallete_colour[7], spr_rend_valid[7] );

always_ff(posedge PPU_SLOW_CLOCK) begin
  case(bkg_draw_state2)
    BUFF_SLICE_1: begin
      chr_ptr_0 = {NAME_TABLE_0[nt_ptr_next],1'b0,tile_row};
      bg_slice_next[15:8] = CHR_ROM_1[chr_ptr_0];
      bkg_draw_state2 <= BUFF_SLICE_2;
    end
    BUFF_SLICE_2: begin
      chr_ptr_1 = {NAME_TABLE_0[nt_ptr_next],1'b1,tile_row};
      bg_slice_next[7:0] = CHR_ROM_1[chr_ptr_1];
      bkg_draw_state2 <= IDLE;
    end
    IDLE: begin
      if(bkg_draw_state == FETCHING)
        bkg_draw_state2 <= BUFF_SLICE_1;
      end
  endcase
endmodule
7 Appendix B - VGA colours
<table>
<thead>
<tr>
<th>Code</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>Color</th>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'h3</td>
<td>3'b010</td>
<td>000</td>
<td>100</td>
<td>#47009F</td>
<td>04</td>
<td>074</td>
</tr>
<tr>
<td>'h6</td>
<td>3'b000</td>
<td>000</td>
<td>101</td>
<td>#A70000</td>
<td>06</td>
<td>0AC</td>
</tr>
<tr>
<td>'h5</td>
<td>3'b101</td>
<td>000</td>
<td>000</td>
<td>#AB0013</td>
<td>05</td>
<td>0B1</td>
</tr>
<tr>
<td>'h18</td>
<td>3'b100</td>
<td>011</td>
<td>000</td>
<td>#8B7300</td>
<td>18</td>
<td>0AF</td>
</tr>
<tr>
<td>'h19</td>
<td>3'b000</td>
<td>100</td>
<td>000</td>
<td>#009700</td>
<td>19</td>
<td>020</td>
</tr>
<tr>
<td>'h1C</td>
<td>3'b000</td>
<td>100</td>
<td>100</td>
<td>#00838B</td>
<td>1C</td>
<td>024</td>
</tr>
<tr>
<td>'h36</td>
<td>3'b111</td>
<td>101</td>
<td>101</td>
<td>#FFBFB3</td>
<td>36</td>
<td>1ED</td>
</tr>
<tr>
<td>'h3C</td>
<td>3'b100</td>
<td>111</td>
<td>111</td>
<td>#9FFFF3</td>
<td>3C</td>
<td>13F</td>
</tr>
<tr>
<td>'h3B</td>
<td>3'b101</td>
<td>111</td>
<td>110</td>
<td>#B3FFCF</td>
<td>3B</td>
<td>17E</td>
</tr>
<tr>
<td>'h38</td>
<td>3'b111</td>
<td>110</td>
<td>100</td>
<td>#FFE7A3</td>
<td>38</td>
<td>1F4</td>
</tr>
<tr>
<td>'h36</td>
<td>3'b111</td>
<td>101</td>
<td>101</td>
<td>#FFBFB3</td>
<td>36</td>
<td>1ED</td>
</tr>
<tr>
<td>'h31</td>
<td>3'b101</td>
<td>110</td>
<td>111</td>
<td>#ABE7FF</td>
<td>31</td>
<td>177</td>
</tr>
<tr>
<td>'h36</td>
<td>3'b111</td>
<td>101</td>
<td>101</td>
<td>#FFBFB3</td>
<td>36</td>
<td>1ED</td>
</tr>
<tr>
<td>'h3B</td>
<td>3'b101</td>
<td>111</td>
<td>110</td>
<td>#B3FFCF</td>
<td>3B</td>
<td>17E</td>
</tr>
<tr>
<td>'h2A</td>
<td>3'b010</td>
<td>110</td>
<td>010</td>
<td>#4FDF4B</td>
<td>2A</td>
<td>B2</td>
</tr>
<tr>
<td>'h28</td>
<td>3'b111</td>
<td>101</td>
<td>010</td>
<td>#F3BF3F</td>
<td>28</td>
<td>1EA</td>
</tr>
<tr>
<td>'h36</td>
<td>3'b111</td>
<td>101</td>
<td>101</td>
<td>#FFBFB3</td>
<td>36</td>
<td>1ED</td>
</tr>
<tr>
<td>'h3C</td>
<td>3'b100</td>
<td>111</td>
<td>111</td>
<td>#9FFFF3</td>
<td>3C</td>
<td>13F</td>
</tr>
<tr>
<td>'h3B</td>
<td>3'b101</td>
<td>111</td>
<td>110</td>
<td>#B3FFCF</td>
<td>3B</td>
<td>17E</td>
</tr>
<tr>
<td>'h28</td>
<td>3'b111</td>
<td>101</td>
<td>010</td>
<td>#F3BF3F</td>
<td>28</td>
<td>1EA</td>
</tr>
</tbody>
</table>

Set Colors
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.

```javascript
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().getActiveSheet().getActiveCell();
  var value = cell.getValue(), color = value;
  cell.setBackground(color);
}
```

8 References

