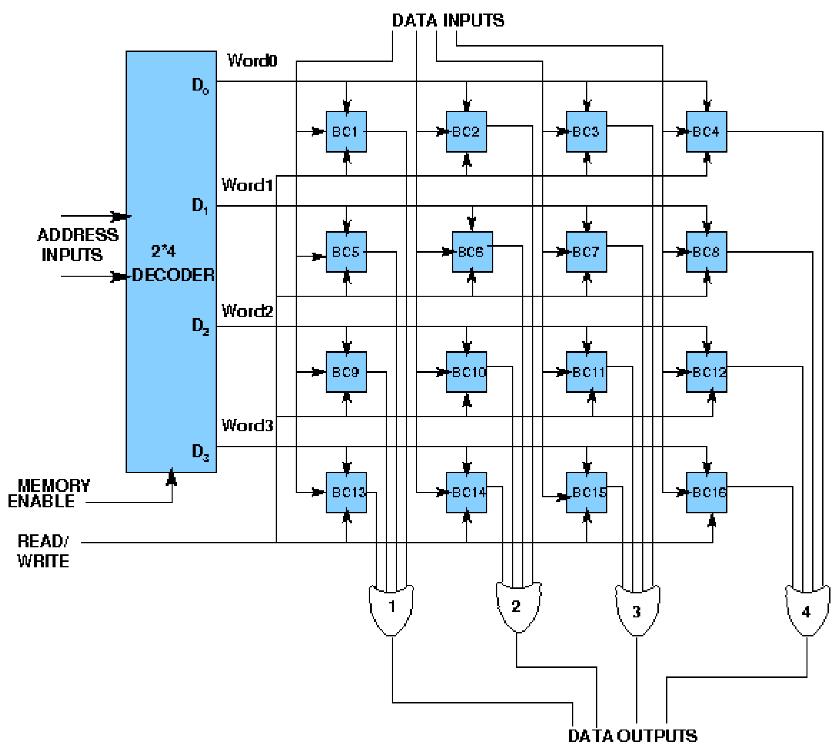
ECE385
DIGITAL SYSTEMS LABORATORY
<u>Introduction to VGA Sprite Drawing</u>

Introduction to Sprites Drawing

The first part of this document discusses some general background information relating to the structure of the font sprite table, which can be found on the course website. The remainder of the document takes a deeper look at how color mapper is used and how to integrate the font sprite table with it. Note that your text mode controller IP will require the text glyphs to be drawn in a 80x30 grid, so that the code provided is not directly usable to you (otherwise there would be no point in the lab). However, the code will give you an idea of how to draw arbitrary graphics starting from the modules provided in the USB+VGA lab (vga.sv and color_mapper.sv) and those concepts will help you design your text mode controller.

RAM Circuitry Overview

We generally expect large amounts of data used in a SystemVerilog program to be stored within the SD RAM. While the provided font table is not stored in the SD RAM, it is designed like one. The following image gives a rough idea of what a simple RAM circuit would look like.



We will input the address into the memory module, and it will output the row of data which the address corresponds to. We are only able to select data at the row level and not the bit level. To select data at the bit level, it will require further processing after the row is obtained.

Font Sprite Table

```
module font_rom ( input [10:0]   addr,
                  output [7:0]   data
                );

   parameter ADDR_WIDTH = 11;
   parameter DATA_WIDTH =  8;
   logic [ADDR_WIDTH-1:0] addr_reg;

   // ROM definition
   parameter [0:2**ADDR_WIDTH-1][DATA_WIDTH-1:0] ROM = {
        8'b00000000, // 0
        8'b00000000, // 1
        8'b00000000, // 2
        8'b00000000, // 3
        8'b00000000, // 4
        8'b00000000, // 5
        8'b00000000, // 6
        8'b00000000, // 7
        8'b00000000, // 8
        8'b00000000, // 9
        8'b00000000, // a
        8'b00000000, // b
        8'b00000000, // c
        8'b00000000, // d
        8'b00000000, // e
        8'b00000000, // f
         // code x01
        8'b00000000, // 0
        8'b00000000, // 1
        8'b01111110, // 2  ******
        8'b10000001, // 3 *      *
        8'b10100101, // 4 * *  * *
        8'b10000001, // 5 *      *
        8'b10000001, // 6 *      *
        8'b10111101, // 7 * **** *
        8'b10011001, // 8 *  **  *
        8'b10000001, // 9 *      *
        8'b10000001, // a *      *
        8'b01111110, // b  ******
        8'b00000000, // c
        8'b00000000, // d
        8'b00000000, // e
        8'b00000000, // f
```

Notice the input and output to the font_rom module. The input is an 11-bit address. The output is an 8-bit chunk of data. As described before, the row number is input as the address, and the entire row is outputted.

By looking at the sprite table, it should become apparent that each symbol takes up 16 rows in the ROM. The general formula for where a symbol resides in the font sprite table would be:

- Starting address: 16*n
- End address: (16*n)+15

where 16 and 15 are in decimal, but n corresponds to the hex code in the table.

We know how to traverse the font sprite table now. Let's put it aside until we review the color mapper.

Color Mapper

Recall the color mapper. There was no reason to edit it for the USB lab, but understanding it will be crucial for implementing more advanced graphics.

```
module  color_mapper ( input        [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
                       output logic [7:0]  Red, Green, Blue );
```

DrawX and DrawY are used to denote which pixel is currently being drawn for the VGA display. Therefore, DrawX can span from 0 to 639, and DrawY will spawn from 0 to 479. Red, Green, and Blue are output to the VGA_Controller to denote the color of the current pixel being drawn.

```
always_comb
begin:Ball_on_proc
    if ( ( DistX*DistX + DistY*DistY) <= (Size * Size) )
        ball_on = 1'b1;
    else
        ball_on = 1'b0;
 end
```

There are two combinational procedure blocks: ball_on_proc and RGB_Display. The ball_on_proc process determines if the pixel currently being drawn lies a certain distance from the center of the ball's center, hence the ball shape. If the current pixel lies within that distance, it is marked as ball_on = 1, otherwise 0. So, the ball_on_proc is simply used to mark the pixels where the ball should be drawn.

```
always_comb
begin:RGB_Display
    if ((ball_on == 1'b1))
    begin
        Red = 8'h00;
        Green = 8'hff;
        Blue = 8'hff;
    end
    else
    begin
        Red = 8'h4f - DrawX[9:3];
        Green = 8'h00;
        Blue = 8'h44;
    end
end
```

The RGB_Display looks at what object the current pixel was marked as, and assigns the RGB color information that should be shown on the screen. All the pixels marked as ball_on = 1 are displayed with a certain color, while the other pixels represent the background of a different color.

The following example shows how to draw two squares on the screen of different colors. There is more than one way to implement this. However, this example should give a strong idea of how one can manage several different objects at once.

First, we will declare 5 logic registers to store data for drawing one shape. These will be:

```
logic shape_on;                 //Used in the same way as ball_on
logic[10:0] shape_x = 300;      //Marks the top left corner of the quadrilateral
logic[10:0] shape_y = 300;
logic[10:0] shape_size_x = 10;  //x and y dimensions of the quadrilateral, in pixels
logic[10:0] shape_size_y = 10;
```

This will allow us to easily draw a rectangle of arbitrary size at arbitrary location. This will later be used as the area for the sprite. Another is declared similarly for the second shape. Assign a location and size of your choosing. Note that for the text mode VGA controller IP, shape_x and shape_y will be at fixed intervals, but this is just an example.

The ball_on_proc and RGB_Display now look like this

```
always_comb
begin:Ball_on_proc
     if(DrawX >= shape_x && DrawX < shape_x + shape_size_x &&
        DrawY >= shape_y && DrawY < shape_y + shape_size_y)
     begin
        shape_on = 1'b1;
        shape2_on = 1'b0;
     end
     else if(DrawX >= shape2_x && DrawX < shape2_x + shape2_size_x &&
             DrawY >= shape2_y && DrawY < shape2_y + shape2_size_y)
     begin
        shape_on = 1'b0;
        shape2_on = 1'b1;
     end
     else
     begin
        shape_on = 1'b0;
        shape2_on = 1'b0;
     end
end

always_comb
begin:RGB_Display
    if ((shape_on == 1'b1))
    begin
        Red = 8'h00;
        Green = 8'hff;
        Blue = 8'hff;
    end
    else if ((shape2_on == 1'b1))
    begin
        Red = 8'hff;
        Green = 8'hff;
        Blue = 8'h00;
    end
    else
    begin
        Red = 8'h4f - DrawX[9:3];
        Green = 8'h00;
        Blue = 8'h44;
    end
end
```

The conditional statements in ball_on_proc are simply used to determine if DrawX and DrawY are within the rectangle's boundaries.

Try running it on the FPGA. There should be two squares of different colors.

Next, to apply the sprite from the font table, we will no longer assign a single color to all the pixels that are marked shape_on. Instead, we will read from the sprite table to determine if there is a 0 or 1 at the location. If it is a 1, then we will assign it the foreground color. Otherwise, it will still retain the background color.

This involves some basic coordinate translation from (DrawX, DrawY) into (Bit #, Address) for accessing the sprite table. See if you can figure out why this transformation makes sense.

(Bit #, Address) = (DrawX – shape_x, DrawY – shape_y + 16*n)

Recall that 16*n is used to mark the starting address for the sprite which we want to draw. n corresponds to the code shown in the font sprite table.

So, we can re-assign the values for our two shapes to reflect two font sprites that are next to each other.

```
logic shape_on;                 //Used in the same way as ball_on
logic[10:0] shape_x = 300;      //Marks the top left corner of the quadrilateral
logic[10:0] shape_y = 300;
logic[10:0] shape_size_x = 8;   //x and y dimensions of the quadrilateral, in pixels
logic[10:0] shape_size_y = 16;
```

Let’s declare an instance of the font sprite table inside color_mapper, along with an address register and a data register.

```
logic [10:0] sprite_addr;
logic [7:0] sprite_data;
font_rom (.addr(sprite_addr), .data(sprite_data));
```

Next, we’ll modify ball_on_proc to assign a different value to the address input of the font sprite table, depending on which shape is being drawn.

```
always_comb
begin:Ball_on_proc
     if(DrawX >= shape_x && DrawX < shape_x + shape_size_x &&
        DrawY >= shape_y && DrawY < shape_y + shape_size_y)
     begin
        shape_on = 1'b1;
        shape2_on = 1'b0;
        sprite_addr = (DrawY-shape_y + 16*'h48);
     end
     else if(DrawX >= shape2_x && DrawX < shape2_x + shape2_size_x &&
             DrawY >= shape2_y && DrawY < shape2_y + shape2_size_y)
     begin
        shape_on = 1'b0;
        shape2_on = 1'b1;
        sprite_addr = (DrawY-shape2_y + 16*'h49);
     end
     else
     begin
        shape_on = 1'b0;
        shape2_on = 1'b0;
        sprite_addr = 10'b0;
     end
end
```

If the font sprite table is not being used, it doesn't really matter what value is being assigned to sprite_addr. Recall that font_rom will output the corresponding sprite_data[7:0]. Next, we will modify the RGB_Display section to read the correct bit from the outputted row, sprite_data.

```
always_comb
begin:RGB_Display
    if ((shape_on == 1'b1) && sprite_data[DrawX - shape_x] == 1'b1)
    begin
        Red = 8'h00;
        Green = 8'hff;
        Blue = 8'hff;
    end
    else if ((shape2_on == 1'b1) && sprite_data[DrawX - shape2_x] == 1'b1)
    begin
        Red = 8'hff;
        Green = 8'hff;
        Blue = 8'h00;
    end
    else
    begin
        Red = 8'h4f - DrawX[9:3];
        Green = 8'h00;
        Blue = 8'h44;
    end
end
```

Compile and program. What do you see?

The final step is up to you, as you will have to generalize the above examples to draw the entire 80x30 grid of text characters. One general solution is to use DrawX and DrawY to look up the proper character (row and column) in the VRAM. Then use the index found in the VRAM to address into font_rom and use the corresponding data from font_rom to draw the 8 pixels (also according to DrawX). Note that the dimensions of the sprite glyph (8x16) and the screen (640x480) and the number of rows and columns (30x80) are all convenient so as to avoid having to use expensive divide and modulo operations on the FPGA logic.