IBM 437 Text Renderer

Experiment #7

Aditya Pola(pola3)

Jonathan Wang(jfwang3)

ECE 385: Digital Systems Laboratory

Lab Section AB1

TA: Peidong Yang

Fall 2025

Motivation

In this lab we implemented a text-rendering system, initially starting off with a monochrome text implementation and then developing into a colorized text implementation. To draw text, we separate the screen into a grid, where each cell can store a character. A character glyph is 16 pixels tall and 8 pixels wide, allowing for a total of 30 characters per row and 80 characters per column on a 640 x 480 display. This allows for a total of 2400 characters to be displayed on the screen at a given time in accordance with the grid. There are a limited set of characters that can be displayed, those of which are part of the IBM-437 text set. We can utilize some form of VRAM to maintain which character is meant to be displayed at each cell and decide what to draw in the cell based on a predetermined font rom. From here, we are able to display text based on the contents of the VRAM. By integrating VRAM with the AXI protocol, we can perform writes and reads via C code, allowing for complex text rendering tasks to be implemented with ease.

The signals sent to draw at any given time are sent via some color mapper, which decides what to draw based on the current character(indicated by vram) and how to draw the character(indicated by font rom). The HDMI is able to map the VGA color signals and map them to HDMI output via differential signals. This interface was utilized in lab 6.2 to implement basic drawing. The main change between the 6.2 code and this version, is that 6.2 a few variables such as ball position and draw position to determine whether to color a pixel, and it occurred continuously rather than in discretized chunks like a grid. As mentioned, we used the same color mapper principle to determine what to color a pixel at any given time, but this was determinants, on color palette control registers, font ram data, and VRAM character data.

We implemented communication via the AXI protocol. In lab 6, we developed a system that allowed MicroBlaze to talk to the MAX3421E chip via the SPI protocol. The keycode harvested from the keyboard was then sent to hardware and was accessible in HDL as a binary keycode input. Instead of directly sending the codes to hardware, the AXI-based protocol in lab 7 writes to a VRAM, which acts as a middleman. The VRAM stores the data used by the IP to write to screen, and the software performs AXI reads/writes to interface and modify with the VRAM, thereby modifying the output. There are advantages and disadvantages to each approach. The IP approach is better abstracted, in that the code is more compartmentalized and detached from users. This makes it more portable in a sense. Additionally, the AXI approach scales very easily, since BRAM logic is simple to expand in this case without encroaching on resource utilization limit. That being said, the AXI approach required much more boilerplate code and complex handshaking procedures compared to the Lab 6 approach. Additionally, there is more latency once we switched to BRAM. The Lab 6 approach is definitely more lightweight compared to the AXI approach in Lab 7.

Preparation

I. Monochrome Text Display

The HDMI Text Mode controller IP is connected to the AXI bus and takes inputs of AXI reset and AXI clock. The IP communicates with Microblaze (the IP is a slave; Microblaze is a master). Microblaze writes to the IP the color and glyph code data. Microblaze also tells which cell in the display it wants the IP to change/display. The IP takes in the desired cell, and calculates the cell's corresponding glyph code and color. The IP then produces an HDMI output, which will be displayed on the display. This process is repeated with Microblaze giving the IP different cells to draw.

The writing to the AXI registers first starts by checking the AWVALID and WVALID signals to ensure that the writing address and data are valid. The write address is latched at the same time you assert the AWREADY and WREADY signals. Those 2 signals indicate that the slave (the HDMI text controller IP) is ready to receive write address and data. This part can be done at the same time or at different times. Once you have all 4 signals, you then start writing in to the register (writing in with write data). Write strobe is also used here to ensure only the desired bytes are written. As the writing is happening, the IP sets the AWREADY and WREADY signals to low to let Microblaze know it is not ready for another writing transaction. As this is happening, the IP (slave) sends a BVALID signal and a BRESP signal to let Microblaze know the status of write transaction (in our case, it is successful). Finally, we set BVALID back to 0 once BREADY is set high (the master has received news of the transaction, stop sending news of the status).

The reading from AXI registers first start by checking the ARVALID signal, which tells the IP that the reading address is valid. The address is then latched at the same time the code asserts the ARREADY signal. This signal indicates that the slave is ready to receive the read address. Once the slave (IP) is ready, it outputs the read data located at the read address. It also sets ARREADY to low at this point to tell the master it cannot do another read operation right now. As this is happening, the IP (slave) sends a RVALID signal and a RRESP signal to let Microblaze know the status of read transaction (in our case, it is successful). Finally, we set RVALID back to 0 once RREADY is set high (the master has received news of the transaction, stop sending news of the status).

```
always_comb begin
    //Calculate x and y coordinate of vram cell
    char_x = DrawX >> 3; //Normalized X coordinate of char we want to draw
    char_y = DrawY >> 4; //Normalized Y coordinate of char we want to draw

    char_idx = char_x + (char_y * CHARS_PER_LINE); //2400 Indices
    vram_addr = char_idx >> 2; //4 characters per VRAM Address
    word_idx = char_idx[1:0]; //Last 2 chars tell us which byte to check MOD 4
end
```

Figure 1: Code Snippet Showcasing Address Calculation for 7.1

Since each glyph is represented by an 8 by 16-pixel cell, we get char_x by dividing the DrawX by 8 so that we know which column the cell we are currently drawing in. We get char_y by dividing DrawY by 16 so that we know which row the cell we are currently drawing in. We then do a conversion of 2d array to a 1d array since char_x and char_y are 2d but font rom is 1d. Hence, the char_idx calculation, which is row major. Once we know the char_idx ,where the cell should be, you can get the vram address by dividing the char_idx by 4 because each vram address holds the data for 4 consecutive characters.

```
logic invert; //Inversion Flag
logic[6:0] glyph_code; //7 Bit Glyph Code

always_comb begin
    case(word_idx)
        0:
        begin
            glyph_code = vram_data[6:0];
            invert = vram_data[7];
        end

        1:
        begin
            glyph_code = vram_data[14:8];
            invert = vram_data[15];
        end

        2:
        begin
            glyph_code = vram_data[22:16];
            invert = vram_data[23];
        end

        3:
        begin
            glyph_code = vram_data[30:24];
            invert = vram_data[31];
        end
    endcase
end
```

Figure 2: Code Snippet of Word Decoding

Once we have the vram address, we can get the vram data (we get the vram address in color mapper and send it to the top level, which uses the vram address to get the vram data and sends it back to the color mapper). Because we have VRAM data, we now need the word_idx because we need to determine which of the 4 characters we want to access. Each character is 8-bit. The most significant bit is the invert bit. The other 7 bits are the character's glyph code. This can be figured out by looking at the last 2 bits of the char_idx.

```
always_comb begin
    glyph_row = DrawY[3:0];//Our row is given by LSBs of DrawY[0-15]
    font_addr = {glyph_code, glyph_row};
end
```

Figure 3: Code Snippet Showcasing Font Address Calculation

Now that we have the glyph code, we can access the correct character data in font ram via bit concatenation. The glyph code will be the upper 7 bits of font rom address because it will decide which character to pick. The lower 3 bits of DrawY will be the lower 3 bits of font ram address because it decides what row of the cell it should currently draw. The lower 3 bits of DrawY correspond to which row in the cell we are currently drawing. So, we need the corresponding row's data for the character picked.

```
//Step 4 - Determine whether Pixel should be On/Off based on Font Data
//DrawX[2:0] retrieves which bit in pixel we are drawing
//Subtract from 7 to account for endianness
logic pixel_on;
always_comb begin
    if(invert)
        pixel_on = ~font_data[7-DrawX[2:0]];
    else
        pixel_on = font_data[7-DrawX[2:0]];
end

//Step 5 - Map Colors based on whether pixel should be on
always_comb begin
if(pixel_on) begin //Character - use foreground color
    Red = colors[27:24];    // FGD_R
    Green = colors[23:20];  // FGD_G
    Blue = colors[19:16];   // FGD_B
end else begin //Background
    Red = colors[11:8];     // BKG_R
    Green = colors[7:4];    // BKG_G
    Blue = colors[3:0];     // BKG_B
end
```

Figure 4: Code Snippet of Drawing Logic

We had one color register that stored the RGB data for 2 different colors (foreground color vs background color). We had a pixel_on variable that would select whether to pick the 1st color (foreground color stored in the upper half bits) or the 2nd color (background color stored in the lower half bits). Pixel_on is first determined by the font data, which decides what color the pixel at the specific row and column of the cell should be. We then check the invert bit to see if we should switch the colors (ex: if font data says use the foreground color, but invert bit is on, then we actually use the background color).

```
always_ff @(posedge axi_aclk) begin
    if(axi_aresetn == 1'b0) begin
        vsync_prev <= 1'b1;  // vsync is normally high
    end else begin
        vsync_prev <= vsync;
    end
end

// Detect negative edge of vsync
assign vsync_negedge = vsync_prev & ~vsync;

// Frame counter using clock enable instead of async clock
always_ff @(posedge axi_aclk) begin
    if(axi_aresetn == 1'b0) begin
        frameCount <= 32'd0;
    end else if(vsync_negedge) begin
        frameCount <= frameCount + 32'd1;
    end else begin
        frameCount <= frameCount;
    end
end
```

Figure 5: Code Snippet of Frame Counter Logic

We have 2 registers for drawX and drawY and we just make it store the values of drawX and drawY, given by the VGA controller. We also get a vsync from the VGA controller, which we used to help calculate the frame count. We first had a variable vsync_prev that stored the value of vsync given by the VGA controller on the previous clock cycle. We then compared the 2 values (vsync and vsync_prev). If v_sync is low and vsync_prev is high, that means that v_sync has reached a falling edge, which means that the VGA controller has just finished drawing the entire screen/display. This means that we have just finished another frame and the frame counter should increase by 1. So, we coded the frame counter so that if it detects a falling edge for vsync, it will be incremented by 1. If no falling edge is detected, the frame counter stays the same, unless reset is hit. If reset is pressed, then the frame counter is set to 0. We store the value of the calculated frame counter into a frame counter register.

The frame counter register, drawX register, and drawY register are stored externally outside of VRAM. If the vram address asks for 1 of those 3 registers, we just access the registers externally.

We do this because we do not want the AXI to write into those registers since their data should be determined by internal hardware, not external hardware.

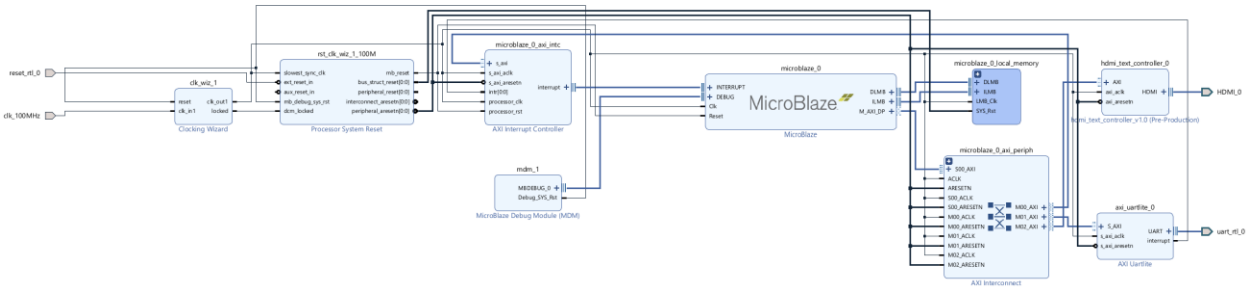

Figure 6: Top-Level Block Diagram for Lab 7.1

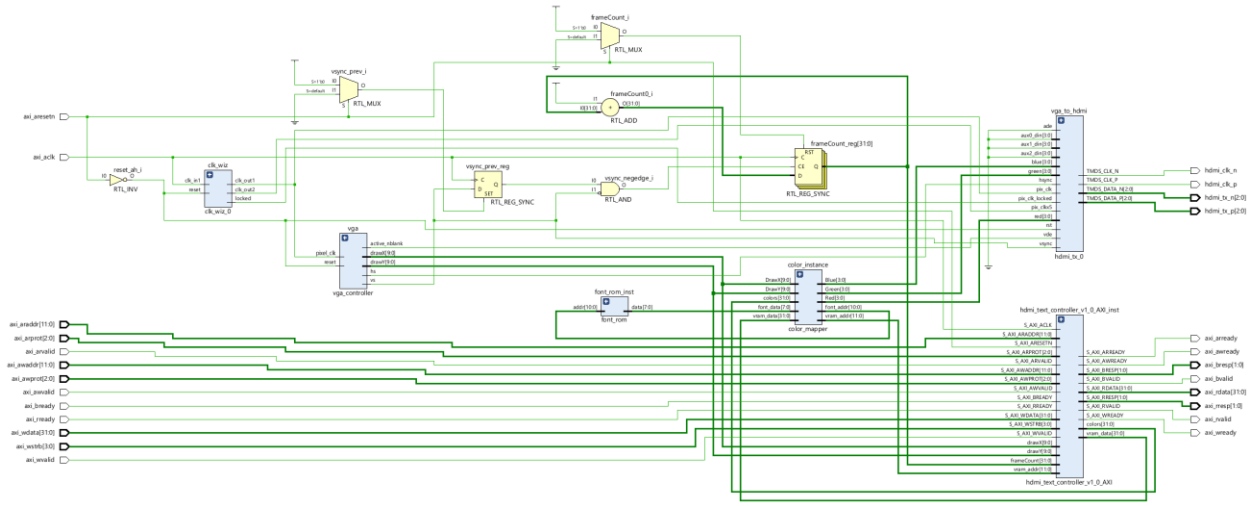

Figure 7: IP Block Diagram for Lab 7.1

II. Color Text Display

For week 2, the first change we made to the IP was we switched from register-based VRAM to on-chip VRAM. We did this because each character data takes twice as much memory space as it used to. Each character now has the invert bit, 7-bit glyph code, 4-bit foreground index, and 4-bit background index. Week 1 p's characters only had the first 2. Our registers were already using almost all of the lookup tables in the FPGA. Week 2 would require us to double those registers, which would not be possible because there simply weren't enough lookup tables. So, we switched to memory-based VRAM. This allowed us to preserve the lookup tables and we had plenty of memory space on the memory-based VRAM. The tradeoff was that accessing the data in memory-based VRAM takes a little longer. Reading takes longer. The handshaking process between master and slave (Microblaze and the HDMI text controller IP) takes an extra clock

cycle because it takes the memory-based VRAM 1 clock cycle delay assert the RVALID signal. The memory-based VRAM has 2 ports: Port A and Port B. Port A is primarily used to write in the glyph codes and the data for the colors into the memory-based VRAM. Port B is for read only, where the color mapper reads what glyph code and its color to print out from the BRAM.
We modified the IP so that instead of 600 registers holding character data, we had 1200 registers holding character data because each character data takes twice as much space as it used to (as described earlier in this section). We now have 8 color registers instead of the 1 in week 1. The color registers, DrawX, DrawY, and framecounter registers were moved to different memory addresses.
The last and most important change to the IP made was modifying the handshaking process for the reads (accessing data from the IP). Since we are now in BRAM, there is a 1 clock cycle latency between the Microblaze asking for the data and the IP sending the data. We changed it so that we added a waiting state where we set on BRAM pending but we haven't loaded the data yet. Then on the following clock cycle, when we check the BRAM pending, it will be on and then the IP will send out the data and set the BRAM pending off. Essentially, every clock cycle the IP is checking BRAM pending and only sends data if the BRAM pending is on. BRAM pending is only on if Microblaze sends an ARVALID signal.

```
always_comb begin
        //Calculate x and y coordinate of vram cell
        char_x = DrawX >> 3; //Normalized X coordinate of char we want to draw
        char_y = DrawY >> 4; //Normalized Y coordinate of char we want to draw

        char_idx = char_x + (char_y * CHARS_PER_LINE); //2400 Indices
        vram_addr = char_idx >> 1; //2 characters per VRAM Address
        word_idx = char_idx[0]; //Last 1 chars tell us which byte to check
end
```

Figure 8: Code Snippet Showcasing Address Calculation for 7.2

Since each glyph is represented by an 8 by 16-pixel cell, we get char_x by dividing the DrawX by 8 so that we know which column the cell we are currently drawing in. We get char_y by dividing DrawY by 16 so that we know which row the cell we are currently drawing in.. We then do a conversion of 2d array to a 1d array since char_x and char_y are 2d but font rom is 1d. Hence, the char_idx calculation, which is row major. Once we know the char_idx ,where the cell should be, you can get the vram address by dividing the char_idx by 2 because each vram address holds the data for 2 consecutive characters.

```
logic invert; //Inversion Flag
logic[6:0] glyph_code; //7 Bit Glyph Code

always_comb begin
    case(word_idx)
        0:
        begin
            glyph_code = vram_data[14:8];
            invert = vram_data[15];
            fgd_idx = vram_data[7:4];
            bkg_idx = vram_data[3:0];
        end

        1:
        begin
            glyph_code = vram_data[30:24];
            invert = vram_data[31];
            fgd_idx = vram_data[23:20];
            bkg_idx = vram_data[19:16];
        end
    endcase
end
```

Figure 9: Code Snippet of Updated Word Decoding

Once we have the vram address, we can get the vram data (we get the vram address in color mapper and send it to the top level, which uses the vram address to get the vram data and sends it back to the color mapper). Because we have VRAM data, we now need the word_idx because we need to determine which of the 2 characters we want to access. Each character is 16-bit. The most significant bit is the invert bit. The next 7 most significant bits are the character's glyph code. The next 4 most significant bits are the fgd_idx (foreground color index). The last 4 bits are the bkg_idx (background color index). This can be figured out by looking at the least significant bit of the char_idx.

```
always_comb begin
    glyph_row = DrawY[3:0];//Our row is given by LSBs of DrawY[0-15]
    font_addr = {glyph_code, glyph_row};
end
```

Figure 10: Code Snippet Showcasing Font Address Calculation

Now that we have the glyph code, we can access the correct character data in font ram via bit concatenation. The glyph code will be the upper 7 bits of font rom address because it will decide which character to pick. The lower 3 bits of DrawY will be the lower 3 bits of font ram address because it decides what row of the cell it should currently draw. The lower 3 bits of DrawY

correspond to which row in the cell we are currently drawing. So, we need the corresponding row's data for the character picked.

```
//Assign color based on foreground and background indices
logic[11:0] fg_color;
logic[11:0] bg_color;

//If index is even, choose lower half of color word
//If index is odd, choose upper half of color word
always_comb begin
  if(fgd_idx[0] == 1'b1) begin
    fg_color = color_palette[fgd_idx[3:1]][27:16];
  end else begin
    fg_color = color_palette[fgd_idx[3:1]][11:0];
  end
end

always_comb begin
  if(bkg_idx[0] == 1'b1) begin
    bg_color = color_palette[bkg_idx[3:1]][27:16];
  end else begin
    bg_color = color_palette[bkg_idx[3:1]][11:0];
  end
end
```

Figure 11: Code Snippet of Color Decoding via Color Palette

We have 8 different colors registers that store 16 different colors (each register stores 2), instead of just the 1 for week 1. You use the 3 three most significant bits of the fgd_idx to determine which of the color register you want to use for the foreground color. You then use the least significant bit of the fgd_idx to determine which of the 2 colors in the color register you wanted to use. If that bit was 1, you pick the color stored in the top half of the register. Else, you pick the color stored in the least significant half of the register. You end up getting 12 bits of data (RGB data) for the foreground color. You repeat the process to get the 12 bits of RGB data for the background color, using bkg_idx this time.

Once you have the foreground and background color RGB data, you do bit concatenation on them to make a 32-bit variable called colors that imitates a color register.

```
//Step 4 - Determine whether Pixel should be On/Off based on Font Data
//DrawX[2:0] retrieves which bit in pixel we are drawing
//Subtract from 7 to account for endianness
logic pixel_on;
always_comb begin
    if(invert)
        pixel_on = ~font_data[7-DrawX[2:0]];
    else
        pixel_on = font_data[7-DrawX[2:0]];
end

//Step 5 - Map Colors based on whether pixel should be on
always_comb begin
if(pixel_on) begin //Character - use foreground color
    Red = colors[27:24];    // FGD_R
    Green = colors[23:20];  // FGD_G
    Blue = colors[19:16];   // FGD_B
end else begin //Background
    Red = colors[11:8];     // BKG_R
    Green = colors[7:4];    // BKG_G
    Blue = colors[3:0];     // BKG_B
end
```

Figure 13: Code Snippet of Update Word Decoding

As we did in week 1, we have a pixel_on variable that would select whether to pick the 1st color (foreground color stored in the upper half bits) or the 2nd color (background color stored in the lower half bits). Pixel_on is first determined by the font data, which decides what color the pixel at the specific row and column of the cell should be. We then check the invert bit to see if we should switch the colors (ex: if font data says use the foreground color, but invert bit is on, then we actually use the background color).

```
if (slv_reg_wren) //Avoid netlist
  begin
    // Handle writes to different regions
    if (axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] >= 12'h800 &&
        axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] <= 12'h807) begin
        // Write to color palette (0x800-0x807)
        for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
            if ( S_AXI_WSTRB[byte_index] == 1 ) begin
              color_palette[axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] - 12'h800][(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            end
    end
  end
```

Figure 14: Code Snippet of Writing to Color Palette

We set the color palette registers by first checking if AXI bus is doing a write. If it is, then we check the vram address. If the address is 1 of the 8-color palettes register address, you write into the color register with the corresponding address. You write the register byte by byte using write strobe. If the write strobe is on, then that byte of the color register would then be replaced by its corresponding byte in the written data on the bus. If the write strobe is off, then that byte of the color register stays the same. You do this for all 4 bytes in the color register.

Color palette registers are stored externally outside of BRAM, because accessing it is simpler with less latency.

We handled the frame counter, drawX, and drawY the same way we did in week 1.

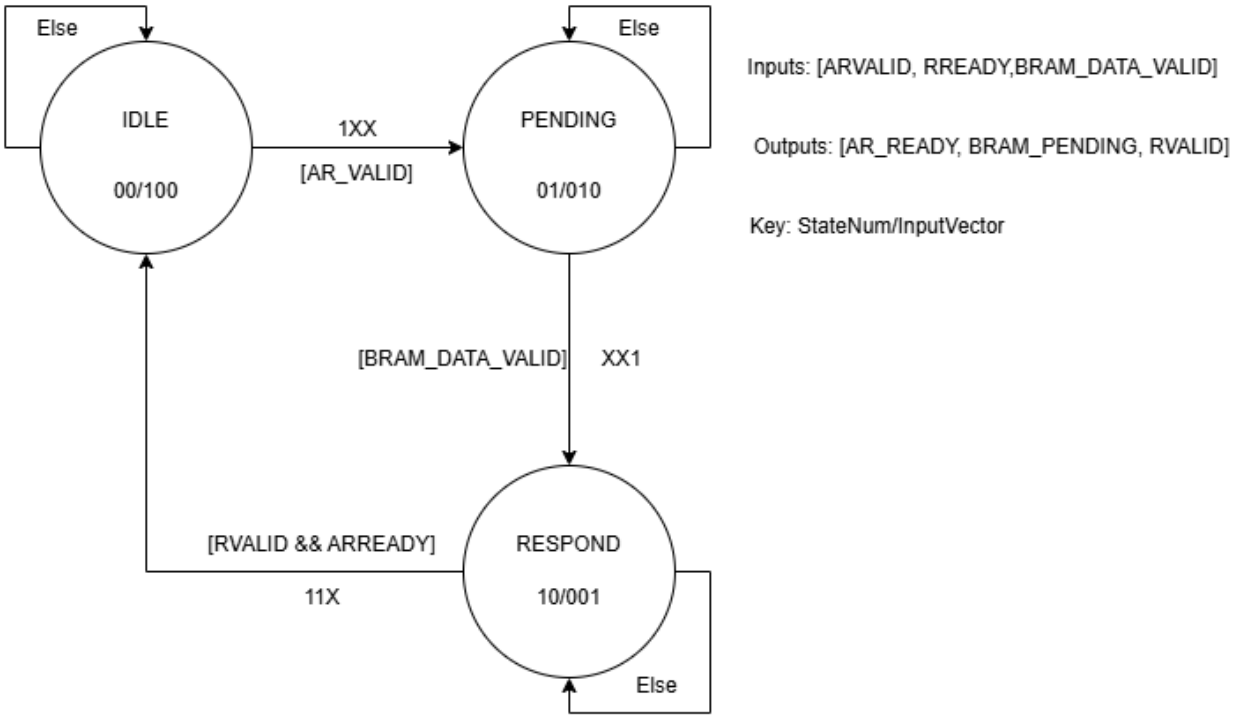

Figure 15: AXI Read FSM Modified for BRAM Latency

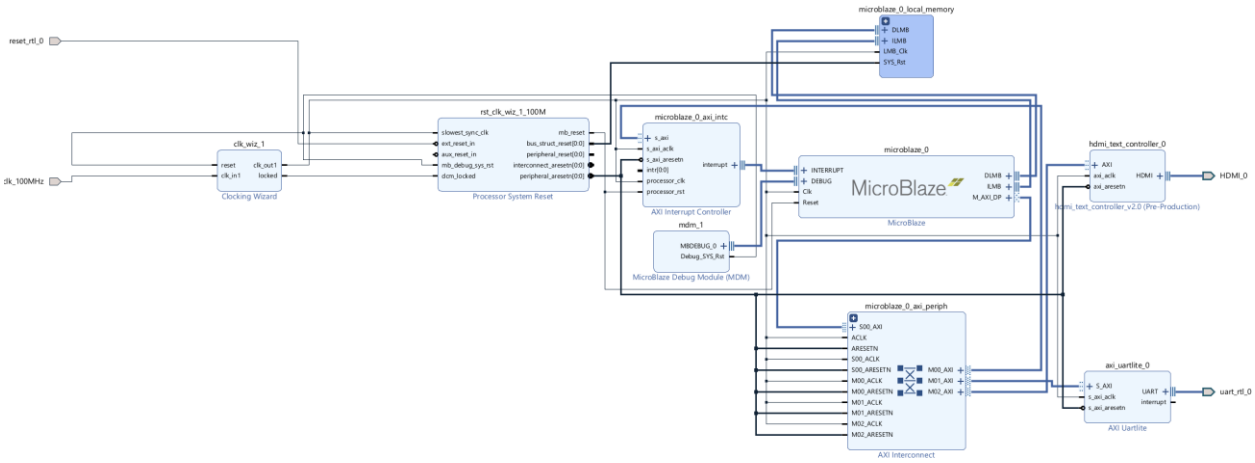

Figure 16: Top-Level Block Diagram for Lab 7.2

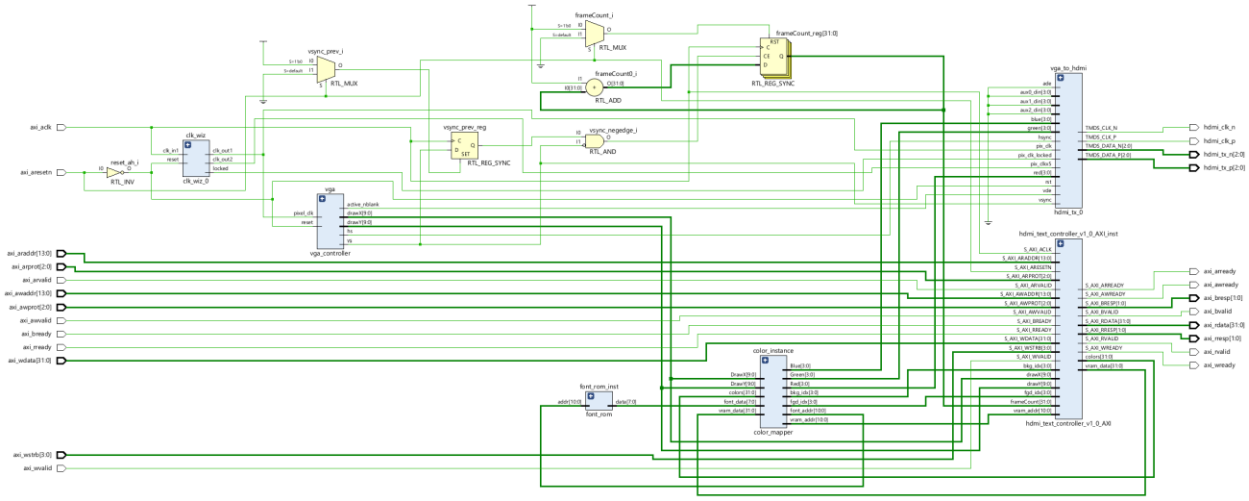

Figure 17: Top-Level Block Diagram for Lab 7.2

III. Code

The following table summarizes the code used to implement the text renderer.

System Verilog File Summaries
Module: hdmi_text_controller_v1_0.sv **Inputs:** axi_aclk, axi_aresetn, axi_awaddr, axi_awprot, axi_awvalid, axi_wdata, axi_wstrb, axi_wvalid, axi_bready, axi_araddr, axi_arprot, axi_arvalid, axi_rready **Outputs:** hdmi_clk_n, hdmi_clk_p, hdmi_tx_n[2:0], hdmi_tx_p[2:0], axi_awready, axi_wready, axi_bresp[1:0], axi_bvalid, axi_arready, axi_rdata[31:0], axi_rresp[1:0], axi_rvalid **Description:** This is the top-level module for the HDMI text controller system. It integrates the AXI4-Lite slave interface, VGA timing generator, font ROM, color mapping logic, and HDMI output IP. The module coordinates the flow of data from the AXI bus (for VRAM and color palette access) through the VGA and color mapping logic to generate the correct pixel data for HDMI video output. It manages synchronization, clocking, and the interface between software-accessible memory and video hardware. **Purpose:** Provide a complete, hardware-accelerated text display system that can be controlled via AXI bus from a processor. It enables dynamic text and color updates in video memory, supporting real-time display changes on an HDMI monitor. **Modifications:** Lab 7.1 [Added necessary files and internal logic to connect files in the top level], Lab 7.2[Added extra logic to route the additional inputs between modules, like color pallete indices or modified address widths]
Module: hdmi_text_controller_v1_0_AXI.sv **Inputs:** S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR, S_AXI_AWPROT, S_AXI_AWVALID, S_AXI_WDATA, S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, S_AXI_ARADDR, S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY, vram_addr[10:0], frameCount[31:0], fgd_idx[3:0], bkg_idx[3:0], drawX[9:0], drawY[9:0]

Outputs: S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP[1:0], S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA[31:0], S_AXI_RRESP[1:0], S_AXI_RVALID, vram_data[31:0], colors[31:0]
Description: Implements an AXI4-Lite slave peripheral for memory-mapped access to the text controller's VRAM, color palette, and control registers. It manages read and write transactions from the AXI bus, decodes addresses, and interfaces with internal dual-port BRAM for VRAM storage. The module also handles color palette updates and provides access to frame count and current draw coordinates for software diagnostics.
Purpose: To allow a processor or other AXI master to read and write text, color, and control data for the HDMI text controller, enabling flexible and efficient updates to the display contents and palette from software.
Modifications: Lab 7.1[Created internal control registers for VRAM via large register network, routed register write operations to corresponding VRAM register or control registers], Lab 7.2[Ported VRAM to be implemented in BRAM, added wait state in FSM and adjusted AXI handshaking protocol times to account for 1 clock cycle latency of BRAM. Also added additional logic to decode color indices sent by color mapper, and return corresponding color data]
Module: vga_controller
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, drawX[9:0], drawY[9:0]
Description: The Vga controller includes the counters needed to keep track of what the current pixel being drawn to is, as well as when frames or lines are finished drawing. It outputs the current drawX and drawY signals for other modules to use, and also outputs the hsync, vsynce, and blanking interval as per the actualy vga control specifications.
Purpose: The Vga controller's purpose is to generate the correct clock signals to keep track of the currently drawn pixel. This allows the ball to know where it is, and where to move after each frame. It also allows the color mapper to know what color to draw any given pixel based on the ball's position along with the current position that is being drawn.

Modifications: None
Module: font_rom.sv
Inputs: addr[10:0]
Outputs: data[7:0]
Description: A read-only memory module containing bitmap data for a set of character glyphs. Each address corresponds to a specific row of a character, allowing the system to retrieve the pixel pattern for any character and row combination.
Purpose: Supply the pixel-level bitmap data needed to render text characters on the display, enabling the text controller to convert character codes into visible glyphs.
Modifications: None
Module: color_mapper.sv
Inputs: DrawX[9:0], DrawY[9:0], font_data[7:0], vram_data[31:0], colors[31:0]
Outputs: font_addr[10:0], vram_addr[10:0], fgd_idx[3:0], bkg_idx[3:0], Red[3:0], Green[3:0], Blue[3:0]
Description: Translates screen coordinates, VRAM data, and font ROM output into RGB color values for each pixel. It determines which character and glyph row to display, selects

foreground and background colors based on VRAM and palette, applies inversion if needed, and outputs the correct color for each pixel.
Purpose: Implement the logic that maps text and color information to actual pixel colors, enabling the display of colored, formatted text on the HDMI output.
Modifications: Lab 7.1[Used different font_data, vram_data, and color addressing scheme to determine the individual colors of characters. Additionally**,** we utilized a 4 character per address addressing scheme to decode characters from vram data], Lab 7.2[Used different addressing scheme to account for 2 characters per address, each with foreground and background indices that must be pulled from color pallete]
Module: hdmi_text_controller_tb.sv
DUT: hdmi_text_controller_v1_0.sv
Description: A comprehensive testbench for the HDMI text controller. It simulates AXI bus transactions to write and read VRAM and palette data, drives the controller through reset and clock cycles, and captures the resulting video output. The testbench can generate a BMP image of the simulated display for verification.
Purpose: Verify the functional correctness of the HDMI text controller design by simulating realistic usage scenarios, checking AXI interface compliance, and validating the visual output.
Modifications: Lab 7.1 [Added AXI Read Test] , Lab 7.2[Modified Testbench to print names in bmp file]

Table 1: File Summaries

Results

```
task axi_read (input logic [31:0] addr, output logic [31:0] data);
    begin
        //STEP 1
        #3 read_addr <= addr;   //Put read address on bus
        read_addr_valid <= 1'b1;    //indicate address is valid
        read_data_ready <= 1'b1;    //indicate ready to receive data


        //STEP 2
        wait(read_addr_ready);//wait for slave ready

        //STEP 3
        @(posedge aclk); //ready and a positive edge

        //At this point, address handshake is complete
        read_addr_valid <= 0; //deassert address valid


        //STEP 4
        wait(read_data_valid); //wait for valid read data


        //STEP 5
        @(posedge aclk); // valid and a positive edge

        data = read_data; //capture read data
        read_data_ready <= 0; //deassert ready for data
    end
endtask;
```

Figure 18: Code Snippet of Modified AXI Read

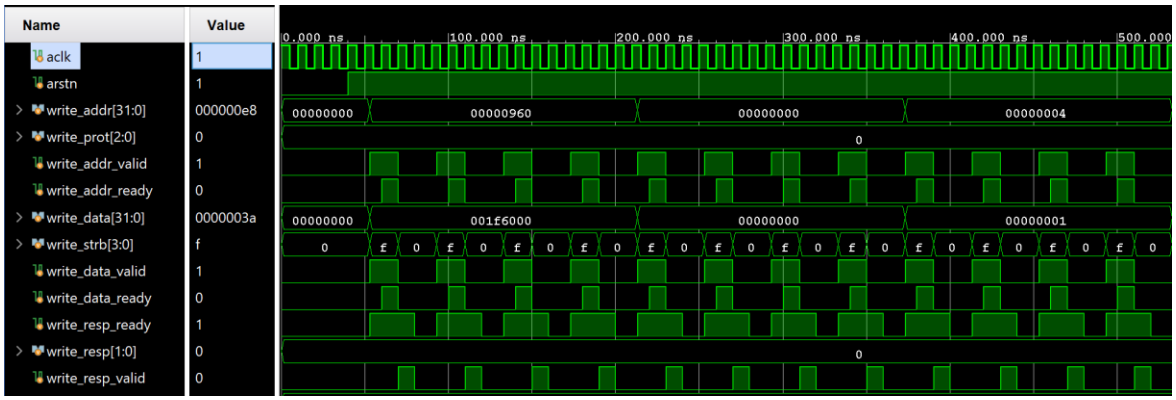

Figure 19: Waveform of AXI Write Transaction

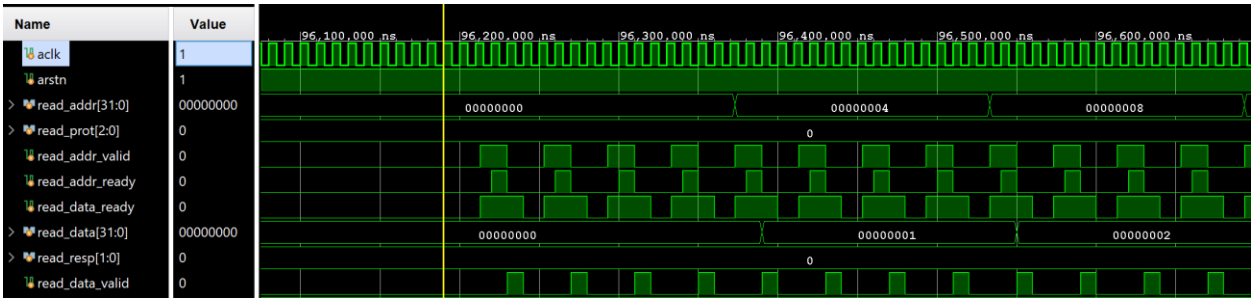

Figure 20: Waveform of AXI Read Transaction

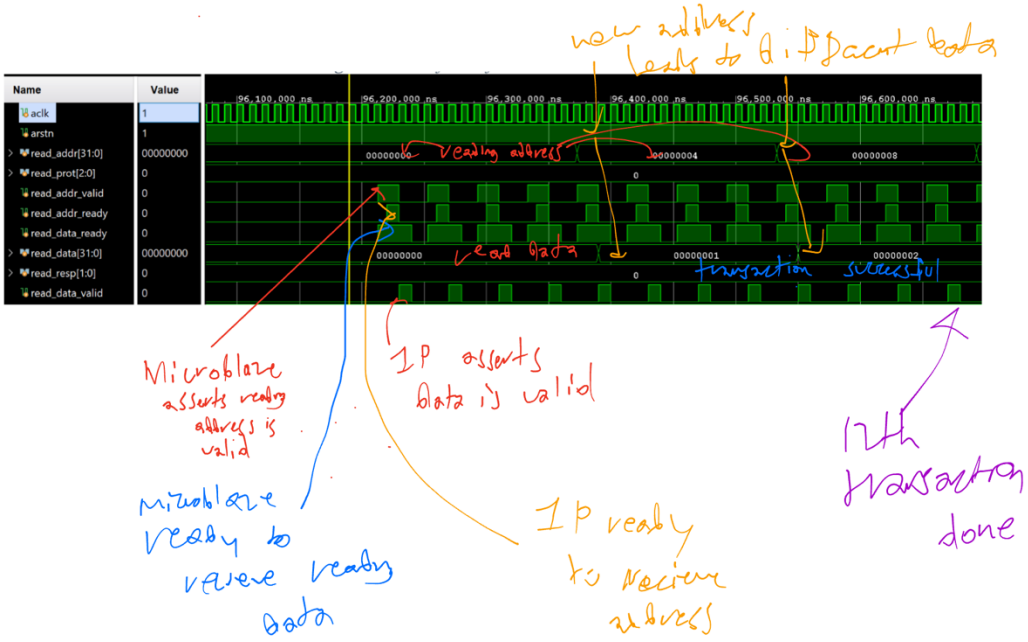

Figure 21: Annotated AXI Write Transaction

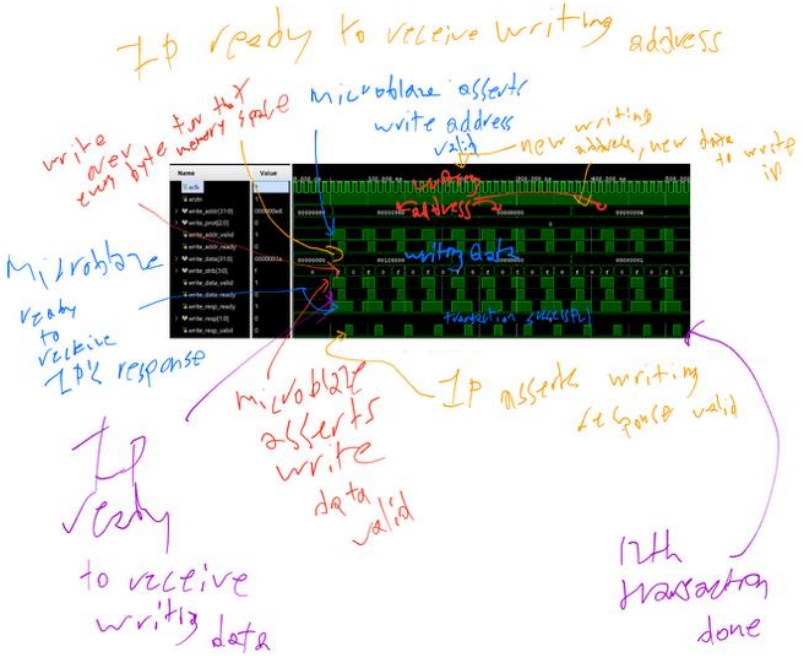

Figure 22: Annotated AXI Read Transaction

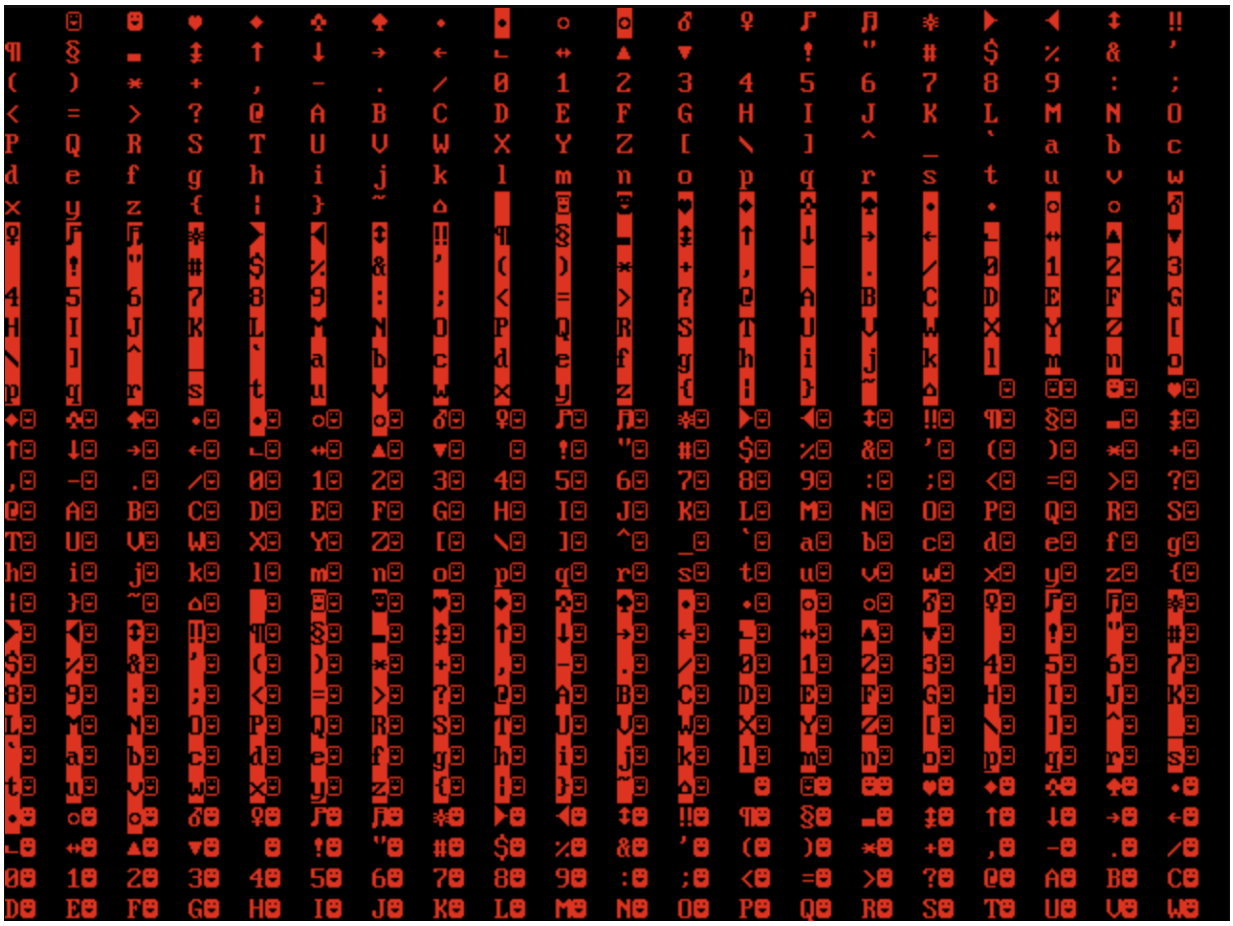

Figure 23: Lab 7.1 Simulated BMP Image

```
//7-1
//Load some random values into VRAM
// for(i=0; i < 600; i++) begin
//   repeat (4) @(posedge aclk) axi_write(4*i, i);
// end

//7-2
//Now we want to load in specific characters into VRAM
//We specifcally want to print 'pola3 and jfwang3 compeleted ECE385!' on the first line
//The glyph codes for these characters(via font rom) are:
//pola3: x70 x6F x6C x61 x33
//' ': x20
//and: x61 x6E x64
//jfwang3: x6A x66 x77 x61 x6E x67 x33
//completed: x63 x6F x6D x70 x6C x65 x74 x65 x64
//ECE385!: x45 x43 x45 x33 x38 x35 x21

//There are 1200 adresses, each storing 2 characters
//We have a sentance of 36 characters -> 18 addresses to fill
//Also note that adresses are litlle-endian
//So for example, to store 'p' and 'o', we need to write 0x6F70 into address 0
//Format is now:
//31 IVR | 30 - 24 Code | 23-20 FGD_IDX | 19 - 16 BGD_IDX
//15 IVR | 14 - 8 Code | 7 - 4 FGD_IDX | 3 - 0 BGD_IDX
//THE color indices correspond to on of the 16 color pallete registers we set above
//Color scheme: pola3 and jfwang3 in Blue on Green, ECE385! in Orange on Green
//Each 32-bit word: {IVR[31], CODE[30:24], FGD[23:20], BGD[19:16], IVR[15], CODE[14:8], FGD[7:4], BGD[3:0]}
//Set some random colors in the pallete
repeat (4) @(posedge aclk) axi_write((2048*4)+0, 32'h00F00F00); // Idx 0=Green, Idx 1=Cyan
repeat (4) @(posedge aclk) axi_write((2049*4)+0, 32'h00700070); // Idx 2=Dark Green, Idx 3=Dark Green
repeat (4) @(posedge aclk) axi_write((2050*4)+0, 32'h0F0000AF); // Idx 4=Orange, Idx 5=Orange
repeat (4) @(posedge aclk) axi_write((2051*4)+0, 32'h00000000); // Idx 6=Black, Idx 7=Black
repeat (4) @(posedge aclk) axi_write((2052*4)+0, 32'h0F0F0FF0); // Idx 8=Yellow, Idx 9=Yellow
repeat (4) @(posedge aclk) axi_write((2053*4)+0, 32'h0F0F0FF0); // Idx 10=Yellow, Idx 11=Yellow
repeat (4) @(posedge aclk) axi_write((2054*4)+0, 32'h00000070); // Idx 12=Dark Blue, Idx 13=Black
repeat (4) @(posedge aclk) axi_write((2055*4)+0, 32'h0FFF0000); // Idx 14=Black, Idx 15=White

//Now write in the characters to VRAM
repeat (4) @(posedge aclk) axi_write(0*4, {1'b0, 7'h6F, 4'h0, 4'h1, 1'b0, 7'h70, 4'h0, 4'h1}); //p o
repeat (4) @(posedge aclk) axi_write(1*4, {1'b0, 7'h61, 4'h0, 4'h1, 1'b0, 7'h6C, 4'h0, 4'h1}); //l a
repeat (4) @(posedge aclk) axi_write(2*4, {1'b0, 7'h20, 4'h0, 4'h1, 1'b0, 7'h33, 4'h0, 4'h1}); //3 ' '
repeat (4) @(posedge aclk) axi_write(3*4, {1'b0, 7'h6E, 4'hF, 4'h1, 1'b0, 7'h61, 4'hF, 4'h1}); //a n
repeat (4) @(posedge aclk) axi_write(4*4, {1'b0, 7'h20, 4'h0, 4'h1, 1'b0, 7'h64, 4'hF, 4'h1}); //d ' '
repeat (4) @(posedge aclk) axi_write(5*4, {1'b0, 7'h66, 4'h0, 4'h1, 1'b0, 7'h6A, 4'h0, 4'h1}); //j f
repeat (4) @(posedge aclk) axi_write(6*4, {1'b0, 7'h61, 4'h0, 4'h1, 1'b0, 7'h77, 4'h0, 4'h1}); //w a
repeat (4) @(posedge aclk) axi_write(7*4, {1'b0, 7'h67, 4'h0, 4'h1, 1'b0, 7'h6E, 4'h0, 4'h1}); //n g
repeat (4) @(posedge aclk) axi_write(8*4, {1'b0, 7'h20, 4'h0, 4'h1, 1'b0, 7'h33, 4'h0, 4'h1}); //3 ' '
repeat (4) @(posedge aclk) axi_write(9*4, {1'b0, 7'h6F, 4'hF, 4'h1, 1'b0, 7'h63, 4'hF, 4'h1}); //c o
repeat (4) @(posedge aclk) axi_write(10*4, {1'b0, 7'h70, 4'hF, 4'h1, 1'b0, 7'h6D, 4'hF, 4'h1}); //m p
repeat (4) @(posedge aclk) axi_write(11*4, {1'b0, 7'h65, 4'hF, 4'h1, 1'b0, 7'h6C, 4'hF, 4'h1}); //l e
repeat (4) @(posedge aclk) axi_write(12*4, {1'b0, 7'h65, 4'hF, 4'h1, 1'b0, 7'h74, 4'hF, 4'h1}); //t e
repeat (4) @(posedge aclk) axi_write(13*4, {1'b0, 7'h20, 4'h0, 4'h1, 1'b0, 7'h64, 4'hF, 4'h1}); //d ' '
repeat (4) @(posedge aclk) axi_write(14*4, {1'b0, 7'h43, 4'h4, 4'h1, 1'b0, 7'h45, 4'h4, 4'h1}); //E C
repeat (4) @(posedge aclk) axi_write(15*4, {1'b0, 7'h33, 4'h4, 4'h1, 1'b0, 7'h45, 4'h4, 4'h1}); //E 3
repeat (4) @(posedge aclk) axi_write(16*4, {1'b0, 7'h35, 4'h4, 4'h1, 1'b0, 7'h38, 4'h4, 4'h1}); //8 5
repeat (4) @(posedge aclk) axi_write(17*4, {1'b0, 7'h20, 4'h0, 4'h1, 1'b0, 7'h21, 4'h4, 4'h1}); //! ' '

//Now just fill in the remaining spaces with the same empty characer withe same background and foreground
for(i=18; i < 1200; i++) begin
    repeat (4) @(posedge aclk) axi_write(4*i, {1'b0, 7'h20, 4'h0, 4'h1, 1'b0, 7'h20, 4'h0, 4'h1}); //' ' ' '
end
```

Figure 24: Code Snippet of Code for Generating 7.2 Sim Image

Figure 25: Lab 7.2 Simulated BMP Image

Figure 26: Screen-Saver Test for Lab 7.2

Post-Lab Analysis

I. Text-Renderer Design Statistics

	Single Color Unit
LUT	14358
DSP	3
Memory(BRAM)	32
Flip-Flop	21092
Latches	0
Frequency (MHz)	Failed Timing(WNS = -2.692)
Static Power (W)	0.074
Dynamic Power (W)	0.407
Total Power (W)	0.481

Table 2: Design Statistics for Single Color Text-Renderer

	Multiple Color Unit
LUT	2503
DSP	3
Memory(BRAM)	34
Flip-Flop	1841
Latches	0
Frequency (MHz)	Failed Timing(WNS = -0.126)

Static Power (W)	0.074
Dynamic Power (W)	0.384
Total Power (W)	0.458

Table 3: Design Statistics for Multi-Color Text-Renderer

Based on Table 2 and Table 3, we can see differences in the two designs. Most obviously, we see there is a huge difference in LUT usage, with the register-based VRAM from 7.1 utilizing 14356 LUTs and 21092 Flip-Flops, while the BRAM based VRAM from 7.2 utilizes 2503 LUTs and 1841 Flip-Flops. Additionally, the design from 7.2 utilizes 2 extra BRAM units. Week 2's design has slightly more power usage but operates faster. Both codes may not be completely optimized, but the BRAM based approach has better WNS. Based on the statistics, I would argue that the BRAM based approach in 7.2 is more efficient as it more effectively utilizes on board resources whilst still allowing for the same functionality(plus additional features like per color text support). Obviously, there are tradeoffs with each design, as the BRAM based approach has additional latency, but takes way less resources. If space is not a priority and the latency cannot be upheld, then a register-based design like 7.1 would be better suited. Alternatively, if latency is not an issue, then the improved space of the 7.2 design makes it much more desirable.

Conclusion

Overall, we were able to implement each week's design properly. We had some minor issues throughout the design process, such as Vitis being buggy, or improperly indexing the color palette register, but these fixes were more or less straightforward to fix. That is, after we recognized what the issue was, we were able to fix it pretty quickly and easily. The instructions provided ample amount of necessary background information to successfully understand and complete each part of the lab. As a result, we don't necessarily think anything needs to be changed with the instructions for following semesters.

For the final project, the AXI protocol, as well as the text renderer, can be incredibly useful for future project designs. If we want to use MicroBlaze and communicate with AXI, we now know the proper handshaking protocols to enable this behavior. Additionally, if we want to display text to screen, whether it be via hardware or through control registers, we now have a color mapper designed to properly render text to certain parts of the screen. We can utilize this for our final project to display text to our screen either through Vitis, or through hardware entirely.