ECE 385

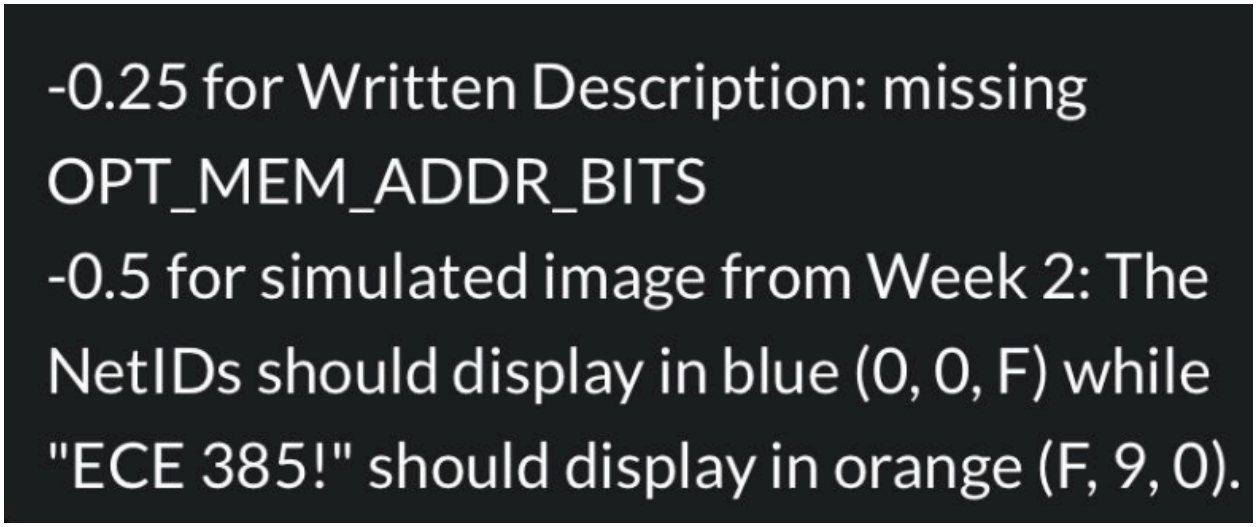
Spring 2024

Experiment 7 - SOC with
Microblaze in SystemVerilog

Lab 7 Report

Ziad AlDohaim (ziada2)
Mohammed Alnemer (mta8)
FOR SH

ECE 385
Lab 7 Report
1. Introduction
2. Written Description of Lab 7 System
a. Week 1 (Monochrome Text Display)
i. Written Description of the entire Lab 7 system
iv. Describe the algorithm used to draw the text characters from the VRAM and font ROM (specifically, describe the equations required to generate the correct addresses to index into the VRAM as well as the font ROM).
v. Describe your implementation of the inverse color bit, as well as the implementation of the control register.
b. Week 2 (Color Text Display) - Hardware Changes for Multi-Color Text
1. Modification from Register-Based to On-Chip Memory-Based VRAM:
2. Modifications to the IP Editor:
3. Modified Sprite Drawing Algorithm:
4. Supporting Multi Colored Text:
5. Hardware/Code for Paletted Colors:
3. Block Diagram
4. Module Descriptions
5. Simulation Waveforms and Images
7. Conclusion



1. Introduction

a. Briefly summarize the operation of the HDMI interface, what are we trying to accomplish this design?

The HDMI is designed to transmit high-definition audio and video signals over a single cable, providing a streamlined connection between various multimedia devices. The HDMI interface's operation within the design focuses on creating a simplified text mode graphics controller connected to the AXI4 memory-mapped bus, supporting 80-column text mode through HDMI output. The primary goal of this design is to enable the display of text characters, initially in monochrome and later in color, on an HDMI-enabled monitor.

b. You should address how the design you created builds on top of the basic one provided for Lab 6.2.

Building from the design in Lab 6.2, this project extends the setup by incorporating a USB interface and IPs to support additional functionalities like USB keyboards and mice. Lab 6.2 laid the groundwork by teaching the basics of system-on-chip (SoC) design with MicroBlaze, including interfacing with peripherals like LEDs and switches and expanding this system with USB and VGA peripherals for a comprehensive multimedia interaction. This design moves from the basic SoC designs to multimedia applications, showcasing the integration of modern interfaces like HDMI and USB to create a functional digital system.

c. In Lab 6.2, we had a different method for hardware->software communication (e.g., the keycode). Describe some advantages/disadvantages of the IP approach

in Lab 7 compared to the approach in Lab 6.2.

In 6.2 the communication method involved direct interfacing with these peripherals, using GPIO for LEDs and switches and specific protocols for USB communication. The hardware-software communication was primarily based on direct reads and writes to specific memory-mapped registers or through simpler communication protocols.

In contrast, Lab 7 introduces an IP core-based approach for the HDMI text mode graphics controller, which is more sophisticated and modular. This IP approach makes the design more scalable and easier to integrate into larger systems. The advantages and disadvantages of moving to an IP-based approach compared to the direct interfacing method used in Lab 6.2 include:

Advantages:

- IP cores can be easily reused in different parts of a design or in different projects. This simplifies the process and reduces time for new projects that require similar functionalities.
- Using IP cores abstracts the underlying implementation details, allowing designers to focus on higher-level system integration without worrying about the intricacies of each peripheral.
- IP cores often adhere to industry standards, ensuring compatibility with a wide range of hardware and software tools, which is crucial for HDMI and other interfaces.

Disadvantages:

- While IPs provide a quick way to integrate complex functionalities, customizing an IP core to specific needs can be more challenging than designing a custom solution from scratch, especially if the IP's internal workings are not fully transparent.
- IP cores might include more functionality than needed for a specific application, leading to unnecessary resource usage or a larger footprint on the FPGA.
- Relying on third-party IP cores introduces a dependency on external sources for updates, support, and compatibility, which could be a concern for long-term maintenance and availability.

All in all, the shift to an IP-based approach in Lab 7 reflects a move towards a more standardized and modular design strategy, offering benefits in terms of design efficiency and ease of integration for complex interfaces like HDMI. However, this approach also requires careful consideration of the potential for increased complexity in customization and dependency on third-party providers.

2. Written Description of Lab 7 System

a. Week 1 (Monochrome Text Display)

i. Written Description of the entire Lab 7 system

The Lab 7.1 system is focused on developing a simplified text mode graphics controller that interfaces with the AXI4 memory-mapped bus to support an 80-column text mode via HDMI output. This system is fundamentally designed to enable the display of text characters on an HDMI-enabled monitor, initially in a monochrome (black and white) format.

The Lab 7.1 system integrates components to achieve its goal of displaying monochrome text through HDMI. At its core is the HDMI Text Mode Controller IP, developed to interface with the system's AXI4 bus. This IP utilizes on-chip memory for video RAM (VRAM) to store text data and a font ROM to store the bitmap representations of each character. The system reads text data from VRAM, retrieves the corresponding character bitmap from the font ROM, and then displays this on the screen via the HDMI interface. This setup allows for the dynamic display of text on an HDMI monitor, controlled through software by writing text data to VRAM.

ii. High-Level Description of the HDMI Text Mode Controller IP

The HDMI Text Mode Controller IP is a digital block designed to generate HDMI signals capable of displaying text on a screen. This IP translates binary text data stored in VRAM into pixel data that represents each character on the screen. The IP is designed around the HDMI specification to ensure compatibility with standard HDMI displays. It operates by reading character codes from VRAM, retrieving the corresponding bitmaps from the font ROM, and then converting these bitmaps into HDMI signals that represent text on the display.

iii. Logic Used to Read and Write HDMI AXI Registers

The HDMI Text Mode Controller interfaces with the system's AXI4 bus, enabling it to communicate with other system components, including the CPU. AXI registers within the controller are used to control its operation, including setting the base address of the VRAM, configuring display parameters, and controlling the display status. The logic for reading and writing to these AXI registers is implemented using standard AXI bus protocols, ensuring smooth data transfer between the CPU and the HDMI controller. Software running on the CPU

can write to these registers to update the display settings or the VRAM contents, and read from these registers to query the status of the controller or the display.

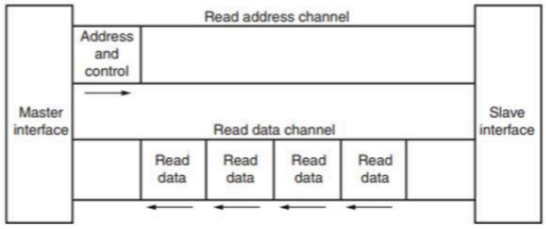


Figure 5 - AXI Read

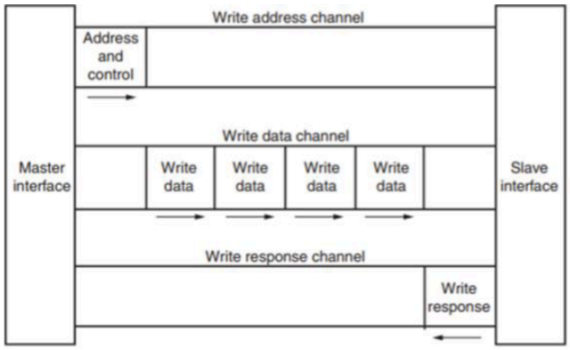


Figure 4 - AXI Write

Data map for AXI read and write transactions

iv. Describe the algorithm used to draw the text characters from the VRAM and font ROM (specifically, describe the equations required to generate the correct addresses to index into the VRAM as well as the font ROM).

The process of displaying text characters involves reading character codes from VRAM, indexing into the font ROM to retrieve the corresponding bitmap for each character, and then drawing each character on the screen. The address in VRAM for a particular character is calculated based on its position in the text grid (e.g., row and column). The formula for computing the VRAM address might look something like VRAM_base_address + (row * row_length + column), where row_length is the number of characters per row. Once the character code is retrieved from VRAM, it serves as an index into the font ROM, where each character's bitmap is stored at a calculated address, typically font_ROM_base_address + (character_code * character_bitmap_size). This bitmap is then used to generate the pixel data for the HDMI output.

v. Describe your implementation of the inverse color bit, as well as the implementation of the control register.

The inverse color bit is a feature implemented within the HDMI Text Mode Controller to invert the color of text for highlighting or other visual effects. This bit can be controlled through a specific AXI register, allowing software to toggle the text color between normal and inverted

modes dynamically. The implementation involves modifying the pixel generation logic to invert the color values when the inverse color bit is set.

b. Week 2 (Color Text Display) - Hardware Changes for Multi-Color Text

The transition to a multi-color text display in the second part of the lab forced us to make several hardware changes. Which aimed at enhancing the video memory and refining the graphics controller to accommodate the expanded requirements for displaying text characters in various colors.

1. Modification from Register-Based to On-Chip Memory-Based VRAM:

To support the additional color capabilities, we transitioned our VRAM from a register-based system to utilizing on-chip memory. This change was crucial to manage the increased data throughput required for multi-color support. The on-chip memory was organized with dual-port functionality, allowing simultaneous read and write operations without conflicts. One port was dedicated to interfacing with the AXI bus for updates, while the other was exclusively used for outputting data to the VGA display, which remained in read mode continually.

2. Modifications to the IP Editor:

In adapting to on-chip memory for VRAM, updates in the IP Editor were necessary. The original 10-bit addressing was expanded to 12 bits to accommodate the enhanced color depth and variety for each character. This expansion allowed us to address a larger memory space, essential for storing the additional color information. The doubling of VRAM capacity required a corresponding increase in the addressability of the memory space by two bits.

3. Modified Sprite Drawing Algorithm:

The sprite drawing algorithm underwent significant revisions to integrate the new on-chip memory approach. The DrawX and DrawY signals were scaled down appropriately to map to the correct memory locations for character retrieval. The VRAM now stored data in 32-bit registers, which included indices for background and foreground colors, character codes, and invert bits. The modulo logic was adapted to handle these new data structures efficiently, ensuring the correct retrieval and display of each character's color attributes.

4. Supporting Multi Colored Text:

Additional changes were essential for implementing multicolored text. The address space within the FPGA needed reconfiguration to manage the new color palette registers effectively. These registers were set up to store 32-bit values, each representing two colors to maximize space efficiency. Logic was added to handle high Address signals differently, interpreting them as accesses to the color palette registers. This setup required detailed management of the address space to ensure correct data retrieval and storage.

5. Hardware/Code for Paletted Colors:

To manage the new palette of colors, the VRAM's capacity was expanded. Each color palette register, accessed through specific addresses, stored comprehensive 32-bit RGB values for the colors. The hardware logic was designed to decode the foreground and background indices stored in VRAM, map these to the appropriate color register, and extract the RGB values for display. This approach required meticulous design to ensure that color assignments were dynamically matched to the text being displayed, allowing for a vibrant and versatile color display.

Microblaze:s

Purpose: Acts as the CPU within Xilinx FPGA systems, executing a wide range of software applications and algorithms.
Operation: It is configurable and can be tailored to specific needs, running everything from simple control tasks to complex processing algorithms directly on the FPGA.

Microblaze Local Memory:

Purpose: Provides fast, directly accessible memory for the Microblaze processor, critical for storing temporary data and instructions.
Operation: Integrated closely with the Microblaze core, it facilitates rapid data access and storage, ensuring high performance of applications by minimizing access delays.

Microblaze Debug Module (MDM):

Purpose: Allows to debug software on the Microblaze processor by providing mechanisms for inspection and control of execution.
Operation: Offers features like breakpoint setting, program stepping, and system reset, allowing for detailed observation and troubleshooting of software behaviors.

Clocking Wizard:

Purpose: Generates and manages the clock signals required for the operation of various components within the FPGA.
Operation: Utilizes external clocks to produce multiple derived clock signals, ensuring that different parts of the system can operate synchronously or at required frequencies.

Processor System Reset:

Purpose: Ensures that all components in the system are correctly initialized and synchronized by managing their reset sequences.
Operation: Can handle multiple reset signals, coordinating the reset process across various modules to ensure a clean start-up or recovery from errors.

AXI Interconnect:

Purpose: Acts as the main artery for communication between the Microblaze processor and peripheral devices, ensuring efficient data transfer and control signal distribution.
Operation: Manages data paths between master and slave devices, allowing for multiple concurrent data transfers and providing flexible bandwidth allocation.

AXI Interrupt Controller:

Purpose: Manages interrupt signals within the system, prioritizing and dispatching them to the Microblaze processor.
Operation: Aggregates multiple interrupt sources, prioritizing them based on system requirements and ensuring that critical interrupts are handled promptly.

AXI Uartlite:

Purpose: Provides a simple serial communication interface for the system, enabling data exchange with external devices or for debugging purposes.
Operation: Implements a UART protocol stack, managing data transmission and reception over serial connections.

AXI GPIO:

Purpose: General-purpose input/output interfaces, allowing the FPGA to interact with a range of external digital devices.
Operation: Can be set up as either input or output ports, enabling bidirectional communication with external hardware like sensors, switches, and LEDs.

Additional Components:

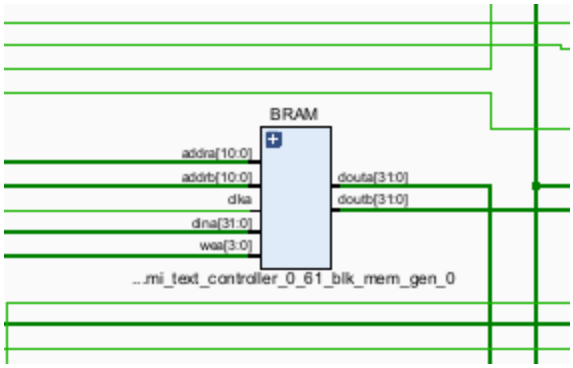
hdmi_text_controller_1_0
Purpose:
The hdmi_text_controller_1_0 module is designed to serve as a simplified text mode graphics controller. It is connected to the AXI4-Lite memory-mapped bus and features HDMI output, tailored to enable the display of text-based graphics on HDMI-compatible devices. The primary components of this module include the Axi Bus Interface, Color_mapper, VGA_controller, Clock Wizard, and VGA to HDMI converter. These integrated modules collectively facilitate the conversion of digital video signals for HDMI output, ensuring compatibility and high-quality display performance.

Operation:
This module leverages VRAM to store display outputs, adapting its architecture to meet different memory requirements across the project's phases. In week 1, VRAM was implemented using registers with color control managed by a dedicated control register, which was just for basic color control. Moving into week 2, the upgraded color differentiation for each character necessitated a shift to using On-Chip-Memory (OCM) instantiated as Block RAM (BRAM). This change accommodated the increased memory demands arising from assigning unique foreground and background colors to each character. Additionally, color management was sophisticated through the implementation of a color palette utilizing eight registers, allowing for more dynamic and varied color presentations in the text display.

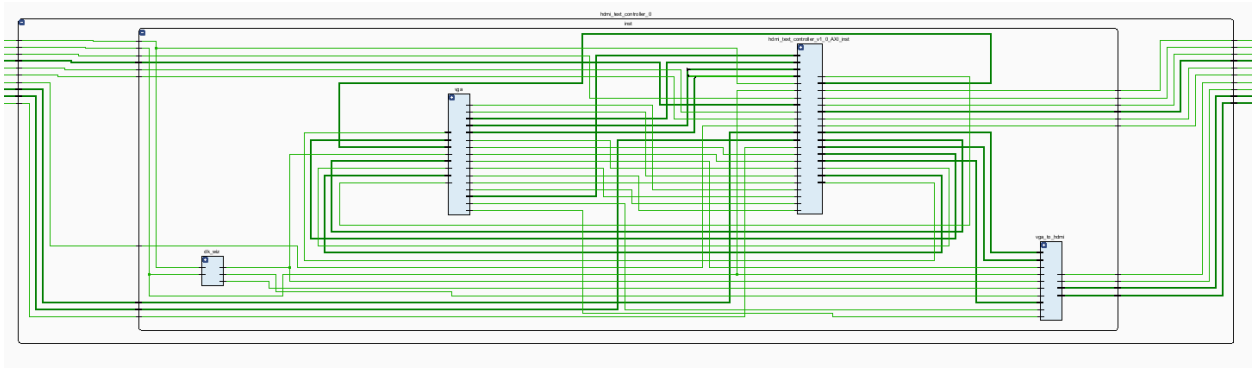
3. Block Diagram

a. This diagram should represent the placement of all your modules in the top level.

WEEK 2:

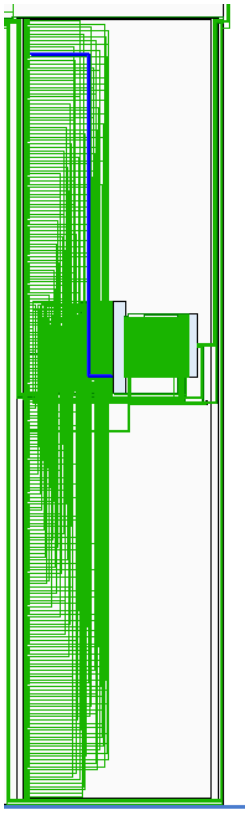


BRAM RTL VIEW

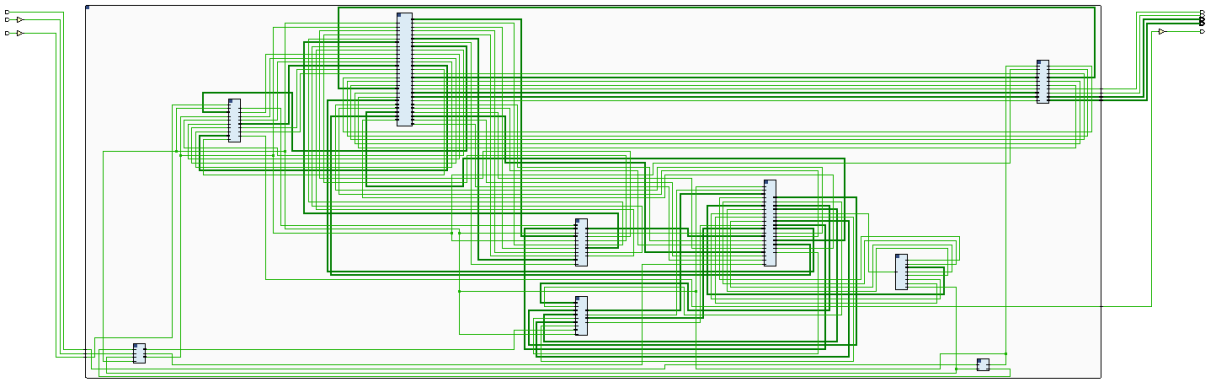


Hdmi_text_controller top view (week 2)

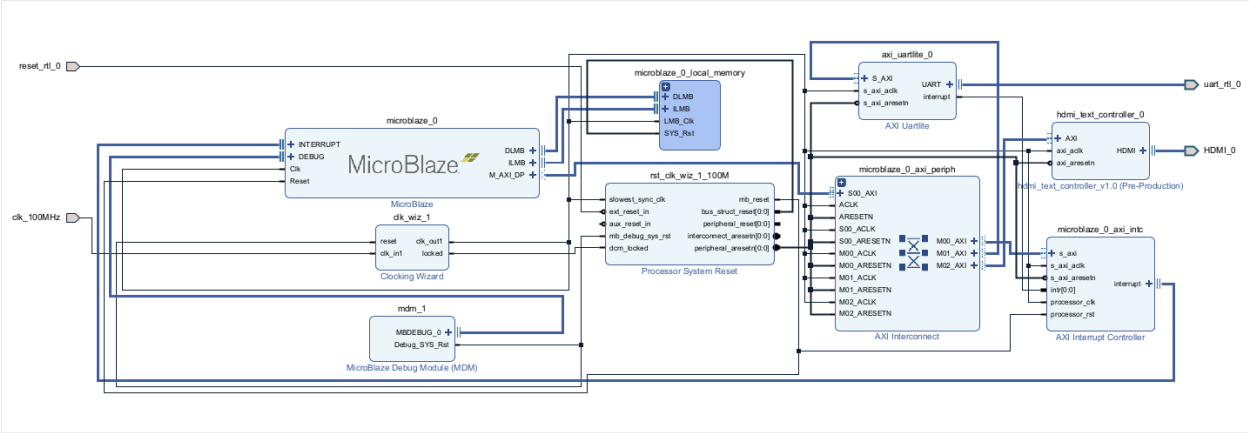
WEEK 1



Hdmi_text_controller week 1 top view



Hdmi_top_level diagram week 1



Block design diagram

b. Internal block diagram of the IP you designed for each week.

i. Note that depending on your layout of the registers inside your main module, the Vivado schematic view may be illegible, in which case you should draw a block diagram using software. You may start with the provided materials (e.g., in IAXI), but you should fill in the specific signals between the modules and the inside subcomponents within each module.
ii. You should have block diagrams for both the Week 1 and Week 2 portions, a good setup is to show the common components (e.g., the SoC setup) first and then show diagrams for both the Week 1 and Week 2 HDMI controller component.

.SV Modules

color_mapper.sv
Inputs:
DrawX[9:0], DrawY[9:0], draw_char[7:0, rgb_values[23:0].
Outputs:
Red[3:0], Green[3:0], Blue[3:0].

Description:
The color_mapper module is responsible for mapping input color data to specific screen pixels based on their coordinates, provided by DrawX and DrawY, and potentially influenced by the draw_char input. This module takes a 24-bit RGB value and processes it to output 4-bit values

for each color component. This processing may involve the reduction of color depth from an 8-bit per channel input to a 4-bit per channel output, suitable for displays or systems that do not support full 8-bit color depth. The module could also implement additional logic to modify the output based on other factors, such as the character being drawn if draw_char is used for such enhancements.

Purpose:
The primary role of the color_mapper module is to translate detailed color specifications into a format suitable for the specific hardware display capabilities. It allows for dynamic color mapping, which is essential for graphic displays that require variable color intensities and effects based on application logic or user interaction. The module is crucial in applications where accurate color representation and efficient mapping to display hardware are required, such as graphical user interfaces, gaming, and other multimedia applications.

Changes:
This module was made from scratch, we took the base from lab 6 and added and removed inputs. For Lab 7.1, we designed new logic to display the monochrome text based on the VRAM design. For Lab 7.2, we changed the logic to accommodate new BRAM.

vga_controller.sv
Inputs:
Pixel_clk, reset
Outputs:
Hs, vs, active_nblank, drawX, drawY

Description:
The vga_controller module is designed to generate synchronization signals and pixel coordinates for a VGA (Video Graphics Array) display. By leveraging the pixel_clk, this module calculates the timing for the horizontal and vertical sync pulses (hs and vs), which are essential for the proper display of images on a VGA monitor. The controller ensures these sync signals are in accordance with VGA timing standards, enabling it to interface with standard VGA displays. The active_nblank output signal distinguishes active drawing from blanking periods, allowing for precise control over the rendering process. The module also calculates the drawX and drawY coordinates for each pixel to be drawn, facilitating the display of graphical content based on input from other system components, like a graphics processor or memory buffer.

Purpose:
The vga_controller module serves as the core interface between a graphical processing unit and a VGA-compatible display device. It manages the critical timing and signal generation required to adhere to the VGA standard, ensuring that the display correctly interprets the data sent from the

host system. This module is vital for applications requiring the output of graphical data to standard monitors, particularly in systems where precise control over display timing is necessary to maintain visual fidelity and performance.

Changes:
This module was given to us

font_rom.sv
Inputs:
Addr[10:0]:
Outputs:
Data[7:0]
Description:
The font_rom module functions as a read-only memory (ROM) storage for font data, specifically designed to store the bitmap representations of characters used in a display system. Each entry in the ROM corresponds to a particular character or a part of a character, depending on the font and character set supported. The 11-bit address input allows for addressing up to 2048 different data entries, which can represent individual characters or rows of pixels in a character bitmap, depending on the system's design.

Purpose:
The primary role of the font_rom module is to provide a reliable and fast method for accessing predefined graphical data, specifically the bitmaps of characters used in text display systems. This module is crucial in systems where text needs to be rendered on graphical displays, serving as a fixed repository of character designs that can be quickly retrieved and used to draw text on the screen. It ensures consistent and efficient access to character data, which is essential for maintaining smooth visual output and system performance.

Changes:
This module was given to us

hdmi_text_controller_v1_0_AXI
Parameters:
C_S_AXI_DATA_WIDTH, C_S_AXI_ADDR_WIDTH

Inputs:
drawX[9:0]: drawY[9:0],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

Outputs:
Draw_char[7:0], rgb_values[23:0], S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA, S_AXI_RRESP, S_AXI_RVALID

Description:
The hdmi_text_controller_v1_0_AXI module is designed to interface with AXI-compliant hardware systems, managing the control logic necessary to display text and graphical data over HDMI. It uses AXI protocol signals to handle data transactions, facilitating communication between the controller and other system components, such as memory and processors. The module utilizes input coordinates (drawX, drawY) to determine where to place characters on the display and outputs the corresponding character data (draw_char) and color data (rgb_values).

Purpose:
This module acts as the central control unit for managing HDMI text display operations in an embedded system. It interprets AXI transactions to fetch or write data relevant to display operations, coordinates the rendering of text at specific screen coordinates, and handles synchronization and reset operations via the AXI interface. The module's purpose is crucial for systems requiring a bridge between microprocessor control and visual output on HDMI displays, particularly in multimedia and user interface applications.

Changes:
Provided file, but modified and extended to meet lab 7. Replaced the VRAM made out of registers to an on chip memory BRAM. Also fixed the read and write logic to accommodate the BRAM.

hdmi_text_controller_v1_0

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,, axi_bvalid, axi_arready ,axi_rdata,axi_rresp, axi_rvalid

Description:
The hdmi_text_controller_v1_0 module serves as a bridge between an AXI interface and an HDMI transmitter. This module facilitates the communication of video data from a system bus (via AXI) to an HDMI interface, which is then transmitted to a display device. It utilizes

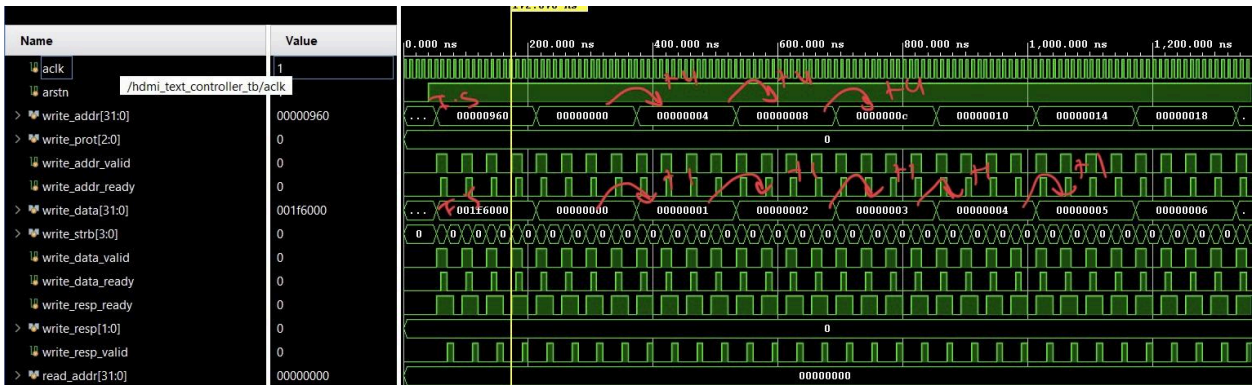
differential signaling for the HDMI clock and data lines to maintain signal integrity over longer distances and higher frequencies, crucial for high-definition video transmission.

Purpose:
This module is designed to handle the complexities of transmitting video data via HDMI, including clock management, data alignment, and interfacing with standard AXI bus protocols. It ensures that video data is processed and transmitted with high fidelity and minimal latency, which is essential for applications requiring high-resolution video output such as video streaming devices, gaming consoles, and multimedia systems.

Changes:
Provided file, but modified and extended to meet lab 7. Just changed some basic input and output changes between colormapper and hdmi_text_controller_v1_0_AXI.

5. Simulation Waveforms and Images

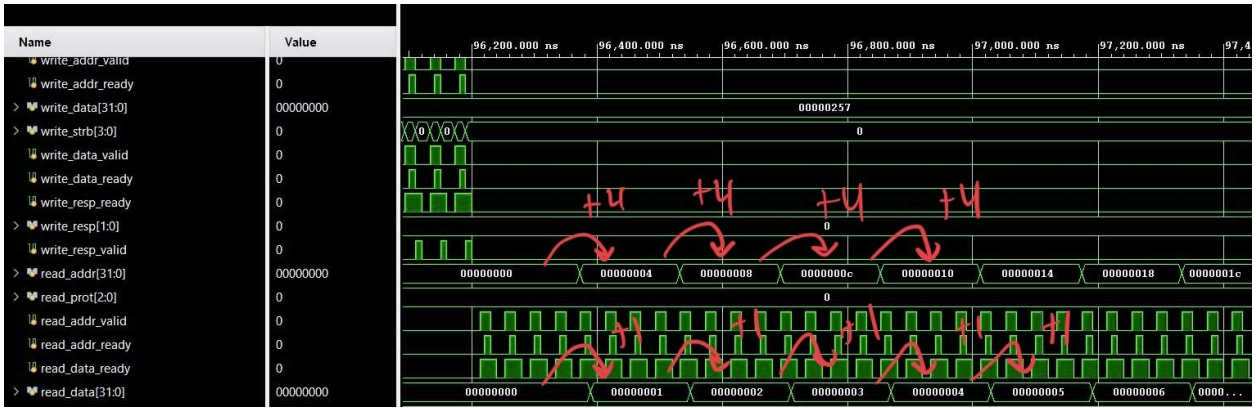
a. Annotated simulation of AXI write transaction (you may use either Week 1 or Week 2 design).



Simulation wave forms displaying the AXI write transaction you can see form the figure the the increment of teh write address "write_addr[31:0]" by four and the increment of the write data "write_data[31:0]" by one. This displays that our AXI write transaction has been implemented successfully.

b. Annotation simulation of AXI read transaction (you may use either Week 1 or

Week 2 design).



Simulation wave forms displaying the AXI read transaction you can see form the figure the the increment of the read address "read_addr[31:0]" by four and the increment of the write data "write_data[31:0]" by one. This displays that our AXI write transaction has been implemented successfully.

Needs review

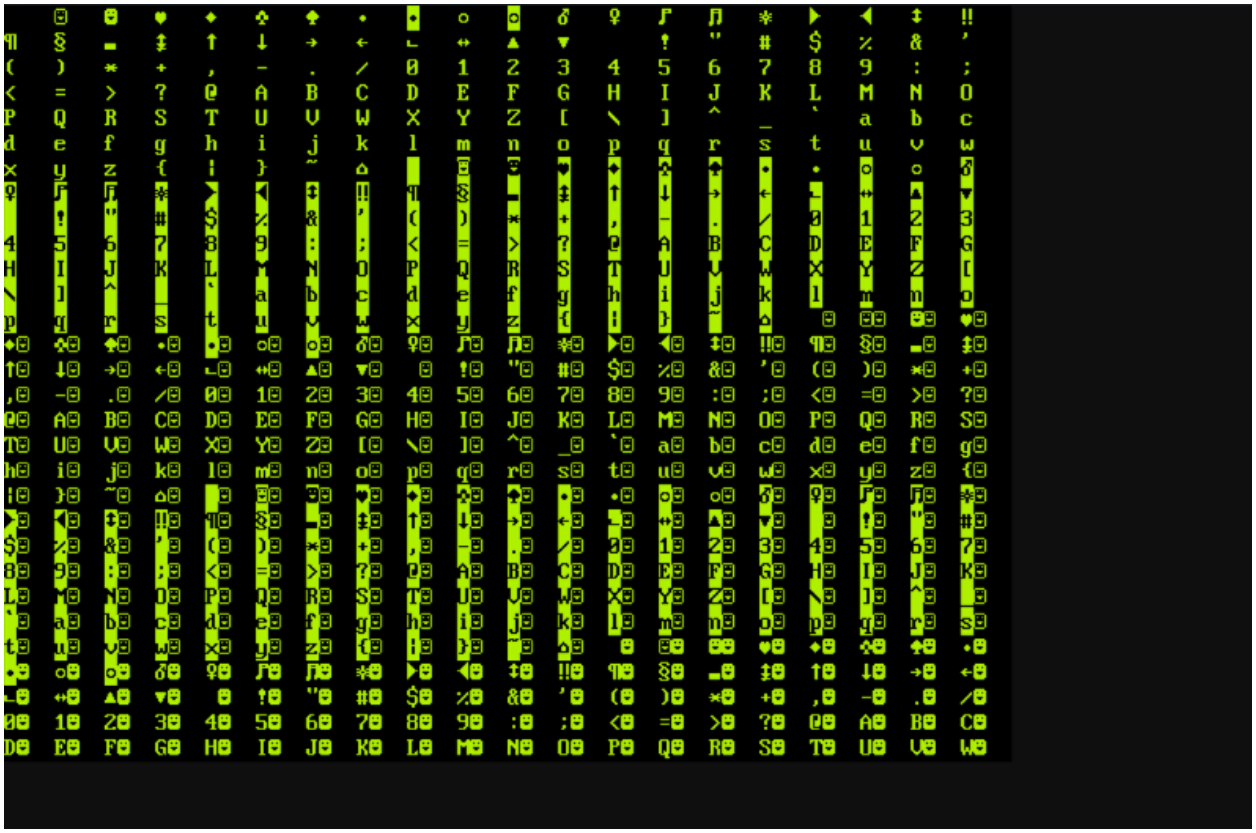
7.5

c. Detailed description / commented code of the axi_read () task you filled out within the provided testbench.

```
230
231         task axi_read (input logic [31:0] addr, output logic [31:0] data);
232             begin
233         // Set the read address
234         read_addr <= addr;
235         // Indicate that the read address is ready to be used
236         read_addr_valid <= 1'b1;
237         //  Indicate that the system is ready to receive data
238         read_data_ready <= 1'b1;
239
240         //  Wait for the device to be ready to send data
241         wait(read_addr_ready);
242         //  Wait for a clock signal to make sure everything is in sync
243         @(posedge aclk);
244
245         // Indicate that the address is no longer needed
246         read_addr_valid <= 1'b0;
247         // Take the data provided by the device and store it
248         data <= read_data;
249         // Wait for the device to confirm that the data is ready to use
250         wait(read_data_valid);
251         //  Wait for another clock signal to finish the read process
252         @(posedge aclk);
253
254         //  Indicate that the data has been received
255         read_data_ready <= 0;
256
257         end
258     endtask;
259
260
```

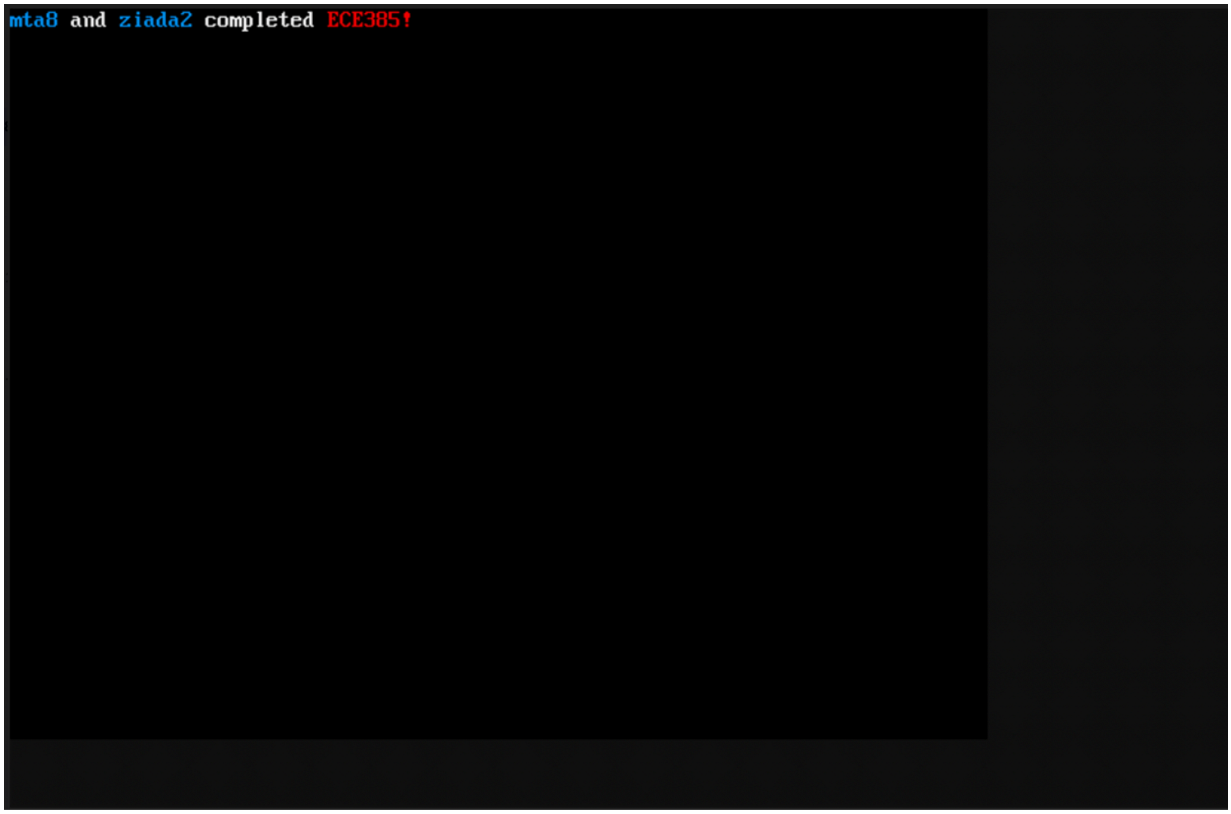
Annotated AXI read code in test bench.

d. Simulated image from Week 1 testbench (see IAMM document, this should not have the red text so it should be generated by you).



Resultant picture from test bench simulation from week 1.

e. Simulated image from Week 2, this text shows the following string “netid1 and netid2 completed ECE 385!”. Fill in your and your partner’s NetIDs. The NetIDs should display in blue (0,0,F) while “ECE 385!



Simulated image from Week 2 displaying netIDs with color change.

f. The commented portion of the modified test bench which calls appropriate AXI writes that display the simulated image as in part 5.e.

```
repeat (4) @(posedge aclk) axi_write(32'h00002000, 32'h00000000);//writing colors into color pallete
repeat (4) @(posedge aclk) axi_write(32'h00002002, 32'h0FF00000);
repeat (4) @ (posedge aclk) axi_write(16'h0000, 32'h74306d30);   //mt   (74 = t) (30 = blue/black) (6d = m) (30 = blue/black)

repeat (4) @ (posedge aclk) axi_write(16'h0004, 32'h38306130);     //a8 (38 = 8) (30 = blue/black) (61 = a) (30 = blue/black)

repeat (4) @ (posedge aclk) axi_write(16'h0008, 32'h6e106110);   //an (38 = n) (10 = white/black) (61 = a) (10 = white/black)

repeat (4) @ (posedge aclk) axi_write(16'h000c, 32'h20006410)      //d (20 = " ") (00 = black/black) (64 = d) (10 = white/black)

repeat (4) @ (posedge aclk) axi_write(16'h0010, 32'h69307a30);  //zi (7a = z) (30 = blue/black) (69 = i) (30 = blue/black)

repeat (4) @ (posedge aclk) axi_write(16'h0014, 32'h64306130);  //ad same logic applies to the rest
repeat (4) @ (posedge aclk) axi_write(16'h0018, 32'h32306130); //a2



repeat (4) @ (posedge aclk) axi_write(16'h001c, 32'h63100000); //c
repeat (4) @ (posedge aclk) axi_write(16'h0020, 32'h6D106F10); //om
repeat (4) @ (posedge aclk) axi_write(16'h0024, 32'h6C107010); //pl
repeat (4) @ (posedge aclk) axi_write(16'h0028, 32'h74106510); //et
repeat (4) @ (posedge aclk) axi_write(16'h002c, 32'h64106510); //ed
repeat (4) @ (posedge aclk) axi_write(16'h0030, 32'h45202000); //E
repeat (4) @ (posedge aclk) axi_write(16'h0034, 32'h45204320); //CE
repeat (4) @ (posedge aclk) axi_write(16'h0038, 32'h33202020); //3
repeat (4) @ (posedge aclk) axi_write(16'h003c, 32'h35203820); //8
```

Annotated code in test bench for writing into memory to display netID colored text

Design Resources and Statistics table

Lab 7.1 Statistics table

LUT	14,971
dsp	3
BRAM	32
Flip-Flops	21,171
Latches	0
Frequency	100.32 MHz
Static Power	0.074 W
Dynamic Power	0.412W
Total Power	0.482 W

Lab 7.2 Statistics table

LUT	2299
dsp	3
BRAM	34
Flip-Flops	1526
Latches	0
Frequency	102.8 MHz
Static Power	0.074 W
Dynamic Power	0.369 W
Total Power	0.443 W

Lab 7.1 - Using Registers: This design uses a lot more Look-Up Tables (LUTs) and flip-flops, which means it uses more of the FPGA's general-purpose resources. It also uses a little less Block RAM (BRAM), which are specialized memory blocks on the FPGA.

Lab 7.2 - Using On-Chip Memory: This design uses fewer LUTs and flip-flops but slightly more BRAM. It's also a bit faster in terms of frequency and uses less power overall.

The design using on-chip memory (Lab 7.2) is more efficient because it uses fewer LUTs and flip-flops. The tradeoff is that it uses more dedicated memory resources (BRAM), but the benefit is seen in better power efficiency and a slight increase in frequency

7. Conclusion

Lab 7 – creating a multi-color text display system – we integrated AXI interface capabilities with HDMI output to display colored text characters. While developing this system, we encountered an issue where the display outputted every other vertical line, which was traced back to an incorrect configuration setting where the blk_hdmi was set to 16 bits instead of the necessary 32 bits. Correcting this to 32 bits resolved the display problems. That was our main issue, however, we also ran into many small bugs throughout the lab, for example, we were printing out only a black box, although we were supposed to be getting the changing text.

This lab taught us skills in managing hardware interfaces and multimedia outputs, which are essential for our Final Project. We could leverage this lab to create an interactive story game that uses dynamic text and visual elements to engage users, building on the text techniques and user interaction methods explored in this and previous labs.

The lab presented significant challenges, particularly due to some ambiguities in the setup instructions related to the blk_hdmi configuration. More detailed setup guidelines and a troubleshooting section could greatly benefit future students by reducing initial hurdles and enhancing the learning experience. Despite these challenges, the lab was well-structured to prepare us for real-world engineering tasks involving system integration and problem-solving. The difficulties we encountered, while frustrating, ultimately contributed to a better understanding of the subject.