ECE 385

Fall 2023

Experiment #7

HDMI Text Mode Controller with AXI4 Interface

Yunxuan Yang, Yuqi Wen
2023/11/12
TA: Tianhao Yu

1. Introduction

a. This design involves creating a graphics controller that supports an 80x30 character display, where each character is drawn from a set of 128 glyphs. The video memory (VRAM) for this display is directly memory-mapped to the AXI4-Lite bus, allowing the MicroBlaze CPU to modify the display content. Additionally, a control register is implemented for screen-wide color settings. The end goal is to render this character-based display and output it through HDMI, providing a basic, yet functional, text-mode graphics system suitable for educational or simple graphical applications.

b. In Lab 7, the design significantly advances from Lab 6.2 by introducing a text mode graphics controller connected to the AXI4 bus with HDMI output, initially in monochrome and later extended to support color. This progression involves a complex redesign around on-chip memory to manage a larger video memory space, a key enhancement over the simpler setup in Lab 6.2. Additionally, the lab emphasizes modifying the device driver for color support and deeper integration of HDMI and VGA controllers, reflecting a more sophisticated approach to hardware-software interaction and graphical output management. The lab concludes with detailed analysis and potential extensions, indicating a comprehensive learning experience in hardware design and system integration.

c. In Lab 7, the use of an Intellectual Property (IP) core approach for the HDMI Text Mode Controller presents a contrast to the simpler hardware-software communication method, like keycodes, used in Lab 6.2. The IP approach offers advantages such as enhanced modularity, enabling easier integration and reuse of the design, and adherence to standardized interfaces like AXI4, which simplifies compatibility with other components. However, it also introduces greater complexity in implementation, potentially higher resource consumption on the FPGA, and a steeper learning curve. While the IP core method provides advanced functionalities, particularly in handling graphics and memory management, it demands a deeper understanding of standard interfaces and more intricate design considerations compared to the more direct and simpler methods used in Lab 6.2.

2. Written Description of Lab 7 System

a. Week 1 (Monochrome Text Display)

i. Written Description of the entire Lab 7 system:

Initially, the lab focuses on creating a monochrome graphics adapter supporting only black and white text. This involves designing an HDMI Text Mode controller IP core, which is functionally similar to the original IBM monochrome graphics adapter. The design includes writing and reading HDMI AXI registers, drawing text characters from the video memory (VRAM) and font ROM, and implementing an inverse color bit and a control register.

In the second week, the lab extends the monochrome text display to support colored text. This requires additional video memory, leading to a redesign of the graphics controller around the limitations of on-chip memory. Key tasks include modifying the register-based VRAM to on-chip memory-based VRAM, updating the IP Editor, altering the sprite drawing algorithm, and implementing hardware/code to draw palette colors.

ii. Describe at a high level your HDMI Text Mode controller IP.

This IP core serves as a bridge between the system's processing unit MicroBlaze CPU and an HDMI output device, translating digital text data into a format suitable for display on an HDMI screen.

iii. Describe the logic used to read and write your HDMI AXI registers.

The logic used to read and write HDMI AXI registers is a crucial part of the HDMI Text Mode Controller IP. This involves setting up the AXI4 'part' of the IP, which defines the interface for the controller. The MicroBlaze processor communicates with the HDMI controller through the AXI4 bus, using specific read and write operations to interact with the HDMI AXI registers. These operations are essential for controlling the behavior of the HDMI output, such as determining what text characters to display and how they are displayed. This process is tested independently of the HDMI drawing portion, ensuring that the register communication is functional before integrating it with the sprite (text glyph) drawing logic. The precise implementation details would depend on the specific AXI4 interface specifications and the design requirements of the HDMI controller.

iv. Describe the algorithm used to draw the text characters from the VRAM and font ROM

We calculate the character's index by multiplying the Cy value by 80, which corresponds to the number of columns in a VGA display, and then adding Cx to it. To find the corresponding character's memory address, we divide the character's index by 4, as each memory address holds four characters. Next, we retrieve the specific character data by utilizing the last two bits of the character's index. Then, to determine the font line address, we multiply the character data by 16 and add the last four bits of DrawY. Using this address, we access the specific line data for

the font. Finally, to extract the pixel data, we subtract the last three bits of DrawX from 7.

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

We extract the inverse bit from the most significant bit of the character data. Should this inverse bit be set to 1, we switch the foreground and background colors; otherwise, we maintain their original colors. The control register, designated as the 601st register, holds color data where bits 1-4 store the background blue value, bits 5-8 for background green, bits 9-12 for background red, while bits 13-16 are used for the foreground blue, bits 17-20 for foreground green, and bits 21-24 for foreground red.

b. Week 2 (Color Text Display)

i. Describe the hardware changes you had to make to support the use of

multi-color text. At the minimum you must describe:

1. Modification of register-based VRAM to on-chip memory-based VRAM. How did your design share the limited on-chip memory ports?

   We apply the true dual port to design the on-chip memory. In this way, we use one port to communicate with axi bus, and one port to communicate with VGA.

2. Corresponding modifications to the IP Editor.

   We replaced the register with BRAM, setting it to true dual port with 1200 depths and 32 bits width.

3. Modified sprite drawing algorithm with the updated indexing equations from on-screen pixels to VRAM.

   Instead of dividing character number by four to get the address number, we divided by two since one word has two bytes in this case. We used the last bit of character number to obtain data from VRAM.

4. Any additional modifications which were necessary to support multicolored text.

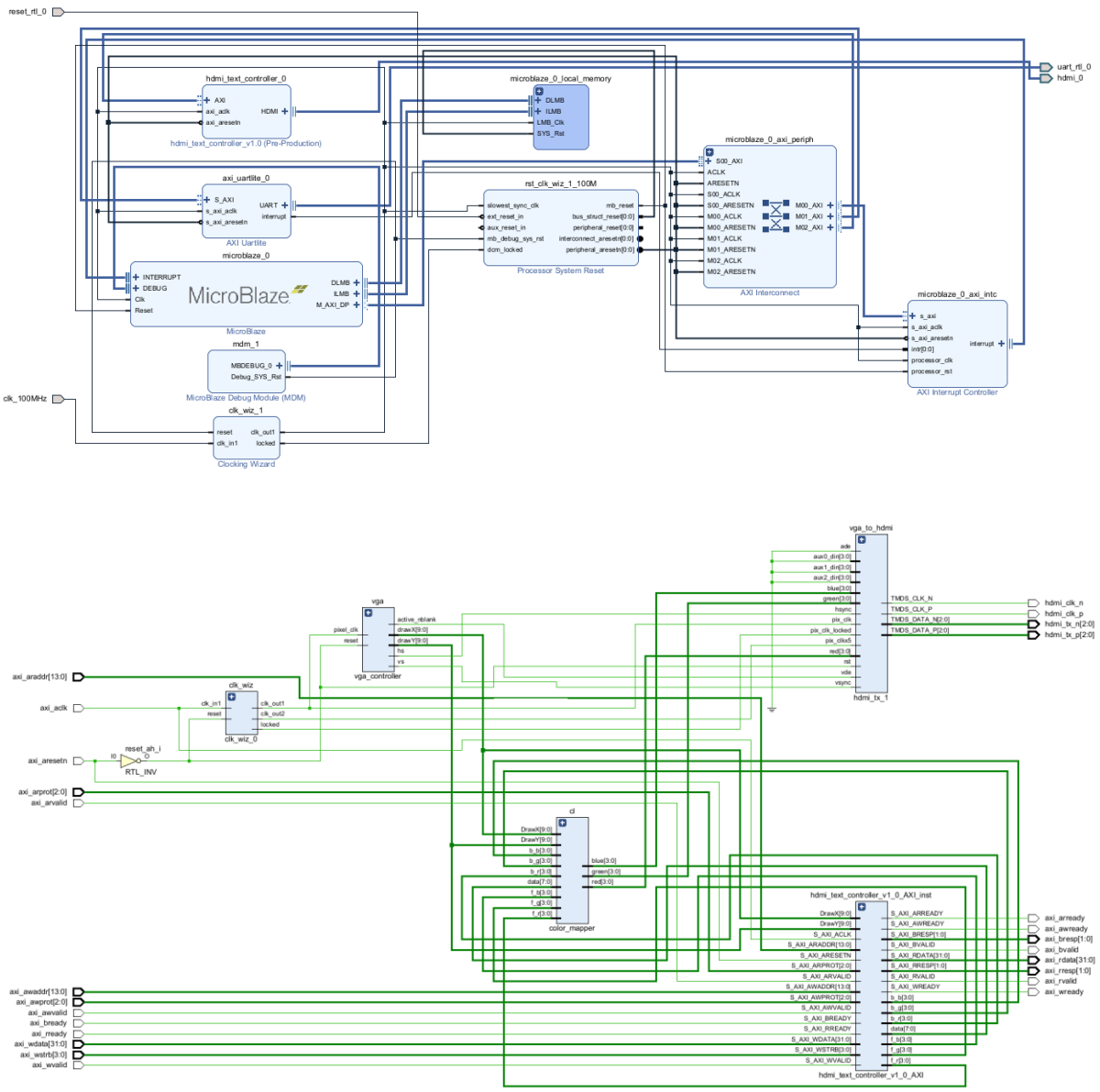
   We checked the address for the axi bus write, if it matched the color palette address, we don’t send the write data to BRAM. Instead, we write it to the color registers.

5. Additional hardware/code to draw paletted colors.

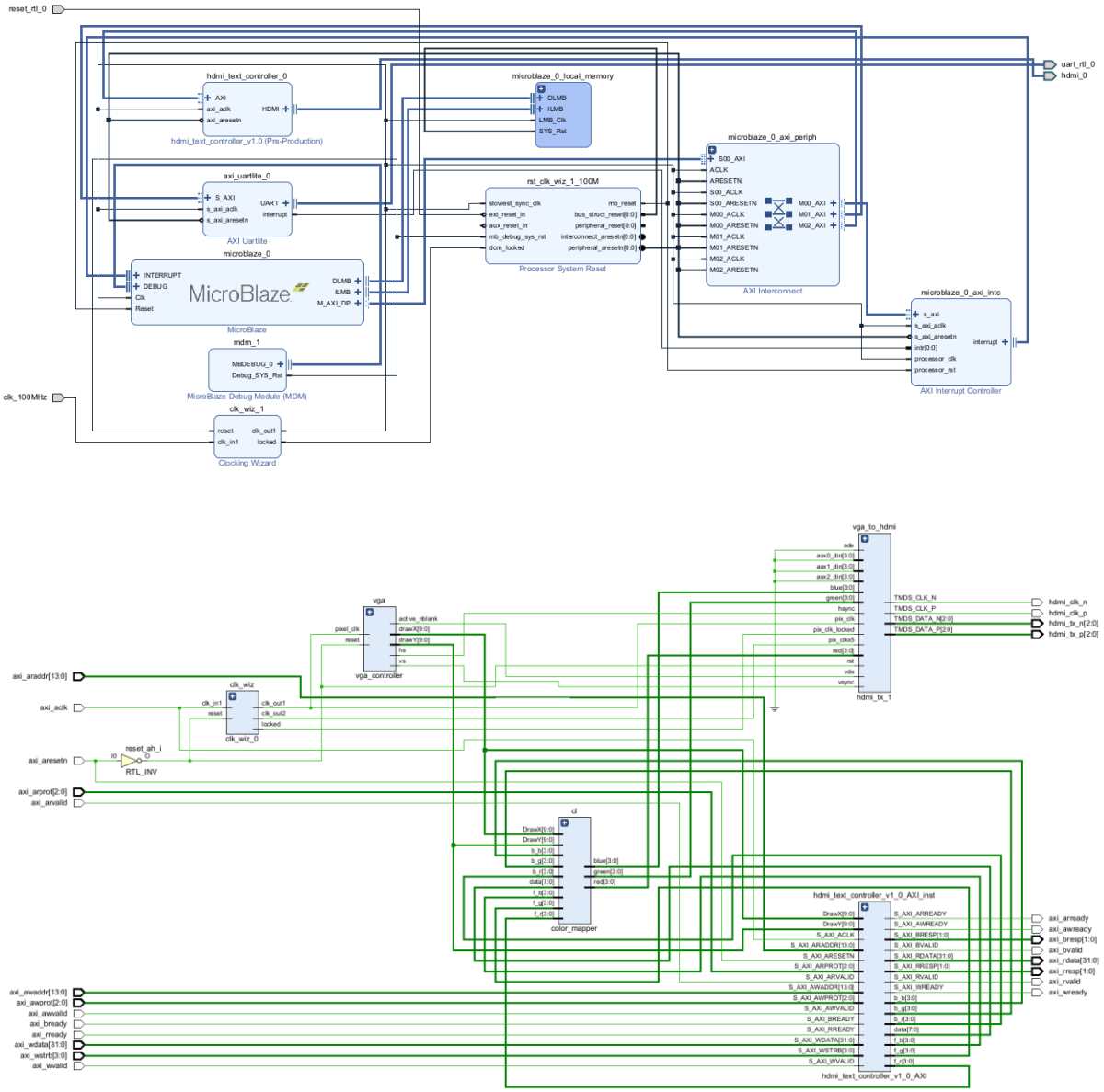
We set up a palette to store multiple colors in an array. We obtain the color by accessing the palette with the color index contained in the character data.

3. Block Diagram

Week 1:



Week 2:



4. Module descriptions:

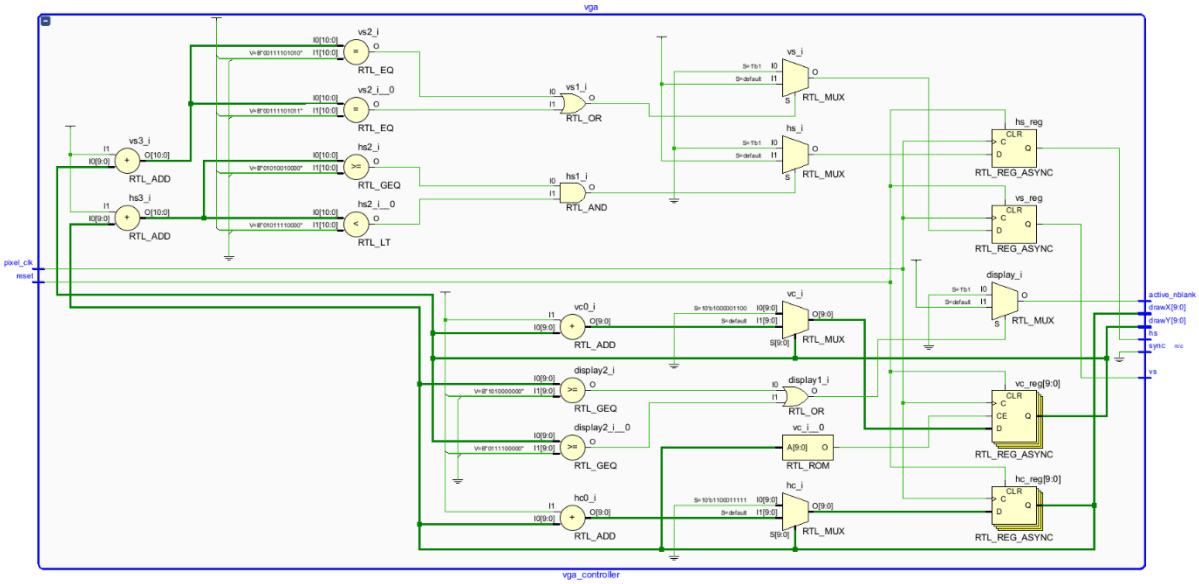
1. Module: VGA_Controller.sv

Inputs: pixel_clk, reset

Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY

Description: This module produces X, Y counters along with H, V sync signals. The blank and sync are left untouched.

Purpose: It's set up to display 640x480 pixels.



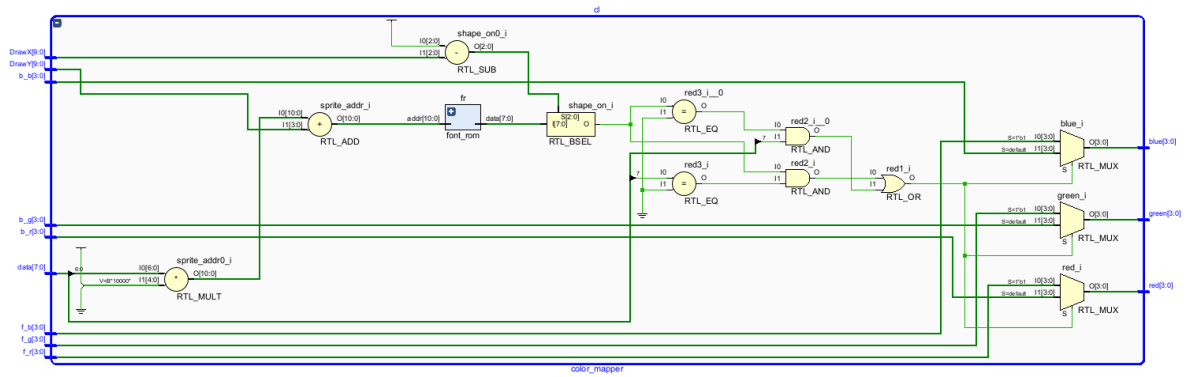
2. Module: Color_Mapper.sv

Inputs: [9:0] DrawX, [9:0] DrawY, [7:0] data, [3:0] f_r,f_g,f_b,b_r,b_g,b_b

Outputs: [3:0] Red, [3:0] Green, [3:0] Blue

Description: Generates (pixelated) character by using the font module.

Purpose: Translates line counters and character data to RGB signals.



3. Module: hdmi_text_controller_v1_0_AXI.sv

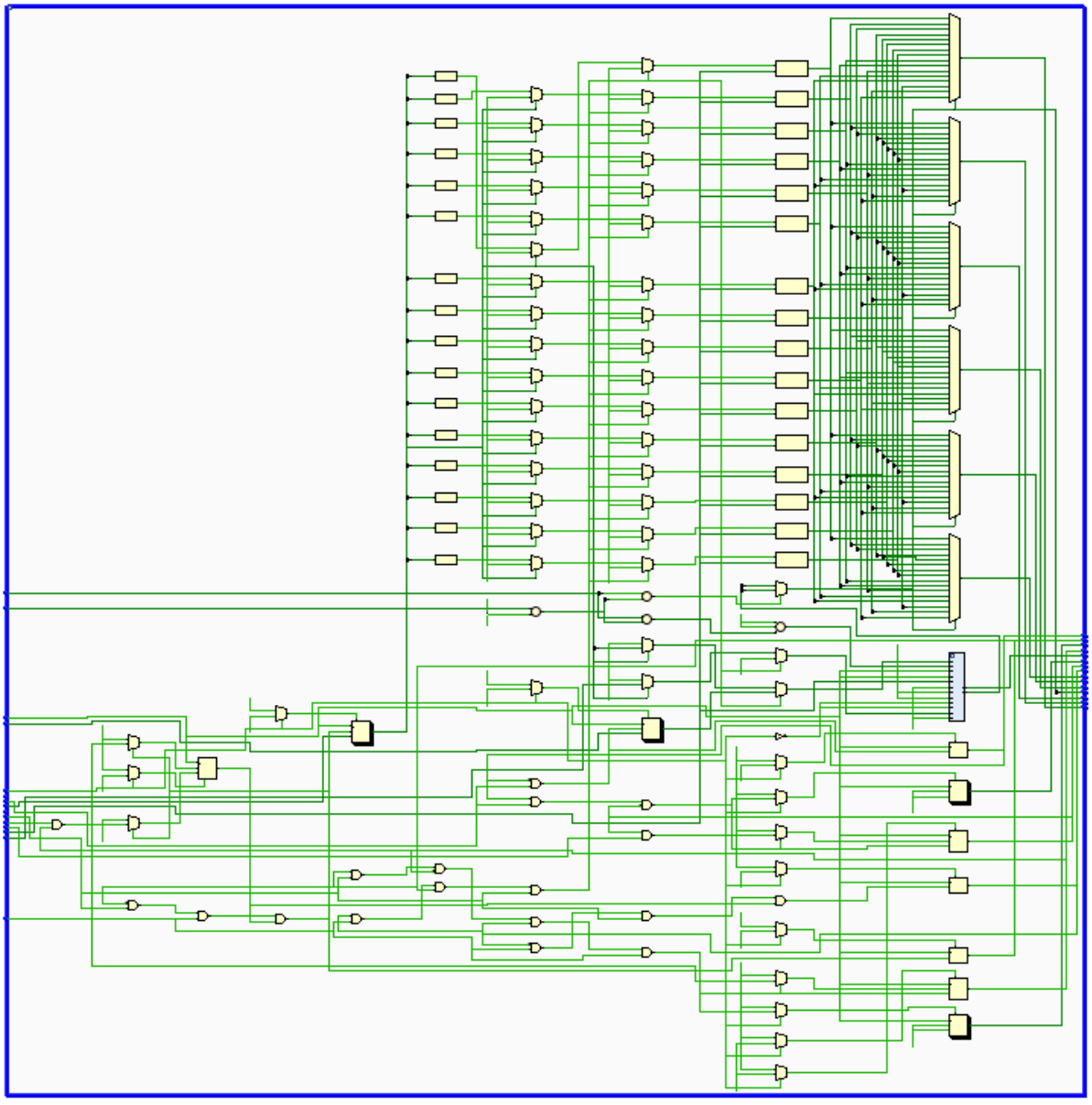
Inputs: [9:0] DrawX, [9:0] DrawY, S_AXI_ACLK, S_AXI_ARESETN, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR, [2 : 0] S_AXI_AWPROT,

S_AXI_AWVALID, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA, [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR, [2 : 0] S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY

Outputs: [7: 0] data, [3:0]f_r,f_g,f_b,b_r,b_g,b_b, S_AXI_AWREADY, S_AXI_WREADY, [1 : 0] S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA, [1 : 0] S_AXI_RRESP, S_AXI_RVALID

Description: The IP of hdmi text controller

Purpose: Process axi bus read write signals, connected to BRAM.



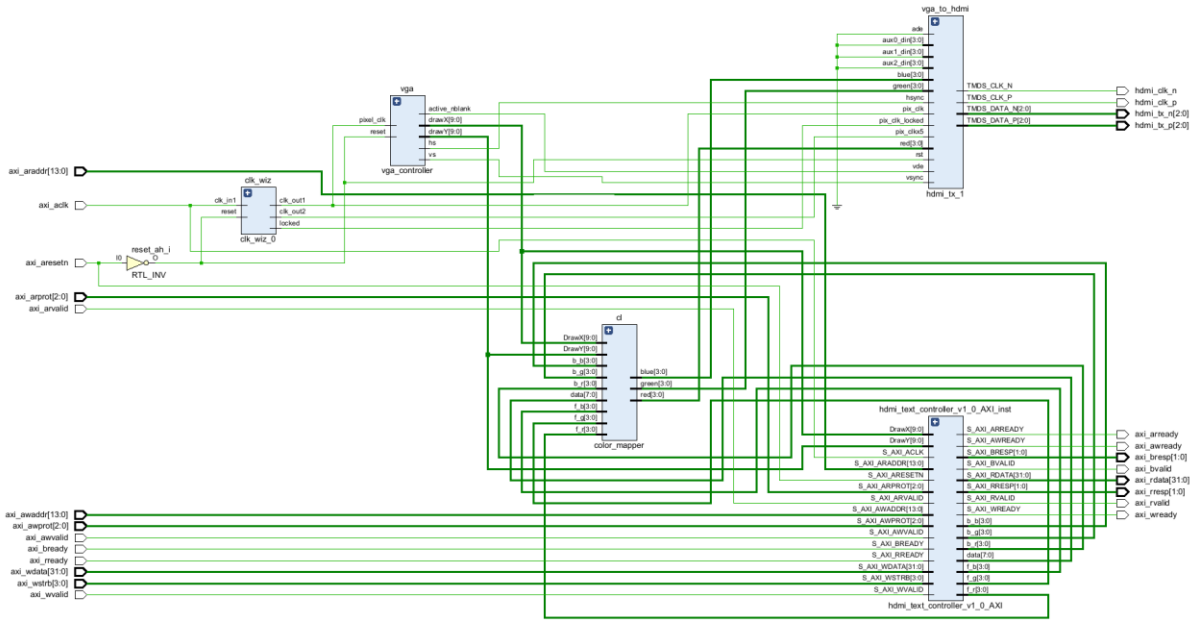
4. Module: hdmi_text_controller_v1_0.sv

Inputs: axi_aclk, axi_aresetn, [C_AXI_ADDR_WIDTH-1 : 0] axi_awaddr, [2 : 0] axi_awprot, axi_awvalid, [C_AXI_DATA_WIDTH-1 : 0] axi_wdata, [(C_AXI_DATA_WIDTH/8)-1 : 0] axi_wstrb, axi_wvalid, axi_bready, [C_AXI_ADDR_WIDTH-1 : 0] axi_araddr, [2 : 0] axi_arprot, axi_arvalid, axi_rready

Outputs: hdmi_clk_n, hdmi_clk_p, [2:0] hdmi_tx_n, [2:0] hdmi_tx_p, axi_awready, axi_wready, [1: 0] axi_bresp, axi_bvalid, axi_arready, [C_AXI_DATA_WIDTH-1: 0] axi_rdata, [1 : 0] axi_rresp, axi_rvalid,

Description: Top level for the IP

Purpose: The module is used as a top-level module to connect each module in this project, receive input signal, compute it with adder.



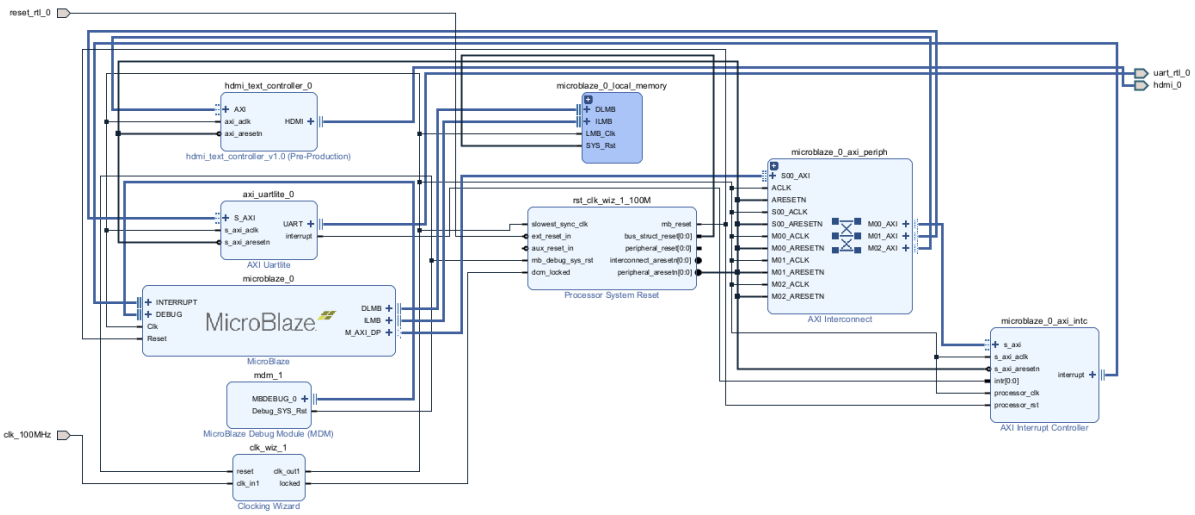
5. Module: top_level.sv

Inputs: Clk, reset_rtl_0, uart_rtl_0_rxd

Outputs: _rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p

Description: Top level for the project

Purpose: The module is used as a top-level module to connect each module in this project, receive input signal, compute it with adder.



Week 2 modification:

6. Module: hdmi_text_controller_v1_0_AXI.sv

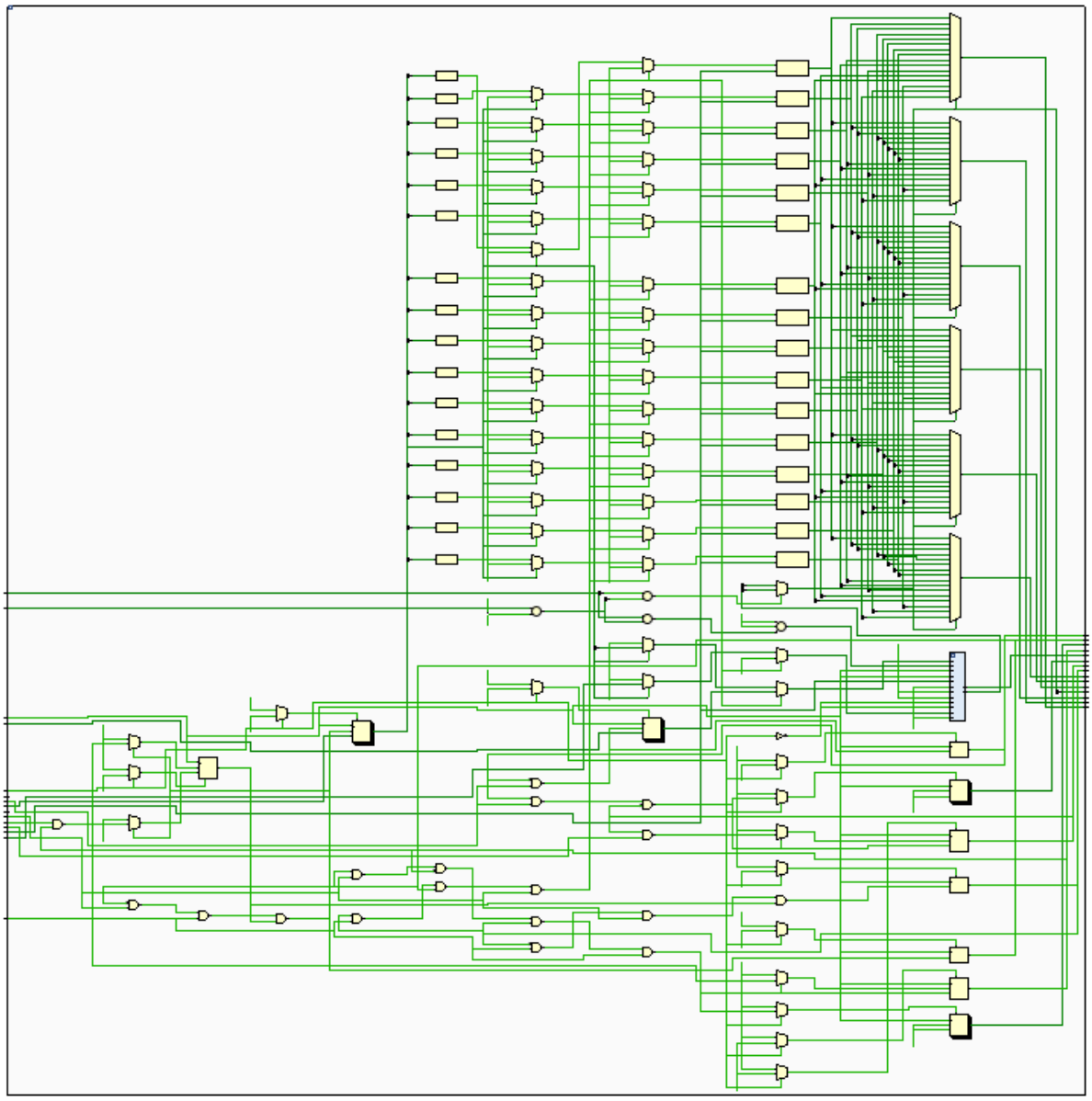
Inputs: [9:0] DrawX, [9:0] DrawY, S_AXI_ACLK, S_AXI_ARESETN, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR, [2 : 0] S_AXI_AWPROT, S_AXI_AWVALID, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA, [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR, [2 : 0] S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY

Outputs: [7: 0] data, [3:0]f_r,f_g,f_b,b_r,b_g,b_b, S_AXI_AWREADY, S_AXI_WREADY, [1 : 0] S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA, [1 : 0] S_AXI_RRESP, S_AXI_RVALID

Description: The IP of hdmi text controller

Purpose: Process axi bus read write signals, connected to BRAM.

Modification: We replaced the registers with the BRAM, created a palette to store colors, and changed the read write processing for the axi bus.



5. Design Resources and Statistics in table

Week 1

LUT	15000
DSP	0
Memory	32
Flip-Flop	21359
Latches	0

Frequency	76.4 MHz
Static Power	74 mW
Dynamic Power	433 mW
Total Power	507 mW

Week 2

LUT	2327
DSP	0
Memory	34
Flip-Flop	172
Latches	0
Frequency	104.9 MHz
Static Power	74 mW
Dynamic Power	396 mW
Total Power	471 mW

VRAM is designed to handle larger amounts of data due to its higher capacity compared to registers. It is suited for storing image and video data that require a lot of space. Registers have very limited storage capacity, suitable for temporary storage of small amounts of data, such as intermediate calculations and control information. While on-chip VRAM is fast, it is generally slower than registers because it is designed to prioritize capacity.

Efficiency:

For high-performance graphics rendering that requires rapid access to a large dataset (like frame buffers in video games or 3D rendering), VRAM is usually more efficient because it can store the necessary data close to the GPU, reducing the need for constant data transfer. For small, fast computations that are critical to performance, using registers can be more efficient as they provide the quickest access time and reduce the latency involved in rendering.

Trade-offs:

Using VRAM comes with the trade-off of potentially higher power consumption and complexity but offers the benefit of larger capacity for graphics-intensive applications. Registers, while providing speed, are limited in capacity and are not suited for storing large amounts of graphical data.

6. Conclusion

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it?

Our design works fine.

b. What are some potential extensions of this design, what did you learn in this lab that might be useful for your Final Project?

We learned that the IP modules like on-chip memory might be helpful in our final project.

c. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

I think the lab manual is ambiguous in the on-chip memory set up part. Lab manual did not tell us how to set the ports and settings.