**ECE 385**

Spring 2024

Experiment 7

# HDMI Text Mode Controller with AXI4 Interface

By Wendi Wang (wendiw2)

## 1. Introduction

### a. The Operation of the HDMI Interface

The HDMI interface, as implemented in the provided code design, serves as a comprehensive platform for generating HDMI signals efficiently. By encapsulating all necessary processes within its architecture, the HDMI interface ensures seamless transmission of signals to the top level. Key components such as clock management, synchronization, and signal conversion are integrated to streamline the signal generation process, optimizing the output for compatibility with HDMI devices.

We are aiming at encapsulating VGA controlling and HDMI signal generating into a single IP, where we can use it for our design as well as adding additional features and functionalities inside this IP.

### b. The Design of the HDMI Interface Based on Lab 6.2

The operation of the HDMI interface is integral to the generation and transmission of HDMI signals. In the provided code design of Lab 6.2, the mb_usb_hdmi_top module encapsulates all necessary processes for generating HDMI signals. By incorporating modules like the VGA controller, clock wizard, VGA to HDMI converter, and color mapper, this interface efficiently handles the complexities involved in signal generation, ensuring high-quality output at the top level.

In the design of the HDMI interface in Lab 7 (based on Lab 6.2), a comprehensive approach is taken to incorporate all relevant modules involved in generating HDMI signals. By assembling modules like the vga_controller, vga_to_hdmi, clock wizard, and color mapper, the HDMI interface is equipped to handle signal generation effectively. Specifically, the VGA controller manages pixel clocking and synchronization, while the VGA to HDMI converter ensures compatibility between VGA and HDMI standards. Additionally, the color mapper facilitates accurate color representation, enhancing the overall visual output. Through the integration of these modules, the HDMI interface based on Lab 6.2 achieves a robust design capable of delivering high-quality HDMI signals.

### c. Advantages and Disadvantages of the IP Approach in Lab 7 Compared to Lab 6.2

Advantages:

- Compared to Lab 6.2 using keycode for hardware->software communication, in Lab 7 we use an IP to make sure everything works well in hardware, then we can communicate with the software part in our design. Its advantage is that we don't need to use software part performance to debug our hardware code, which in turn enhances the reusability and efficiency of our design.
- It enhances the reusability of IP components across various projects. By breaking down functionalities into modular components, each component can be easily reused in different designs, promoting efficiency, and reducing development time.

- The modular nature of IP components provides flexibility for users to add features and functionalities as needed. Since each component is designed to be self-contained and independent, users can customize and extend functionalities without affecting other parts of the design. This flexibility empowers users to tailor the IP to meet specific project requirements, enhancing versatility and adaptability.

Disadvantages:

- When attempting to implement minor modifications or adjustments to the IP components. In such cases, even seemingly small changes may necessitate the repackaging of the entire IP, which can be cumbersome and time-consuming.
- This necessity for repackaging arises due to the interconnected nature of IP components within the design. Since components are tightly integrated and dependent on each other, any modifications to one component may have ripple effects on others.

## 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 system comprises a simplified text mode graphics controller designed for week 1. This controller is interconnected with the AXI4-Lite memory-mapped bus and facilitates an 80-column text mode via HDMI output. It offers support for 80 columns by 30 rows, constituting 2400 characters in total. Each character selection from a subset of IBM codepage 437, represented by 7 bits, with an additional bit enabling the inversion of colors, necessitating 8 bits per character. The font data for each glyph is embedded in the font_rom.sv file, with each character occupying an 8x16 pixel space, resulting in a screen resolution of 640x480, akin to the VGA standard from previous labs.

#### ii. High Level Description of HDMI Text Mode Controller IP

The HDMI Text Mode Controller IP is a pivotal component within the Lab 7 system, responsible for generating and displaying text-based content on an HDMI output. This IP module interfaces with the AXI4-Lite memory-mapped bus to access video memory (VRAM), where text data is stored, enabling seamless integration with the system's MicroBlaze CPU.

#### iii. Read & Write Logic of HDMI AXI registers.

Both Read and Write Logic of HDMI AXI registers are based on hand-shaking protocol.

The read and write logic of HDMI AXI registers adheres to a well-defined sequence for AXI4-Lite transactions. In a read transaction, the Master initiates by asserting ARVALID for a valid address and RREADY to indicate readiness to receive data. The Slave, in turn, confirms readiness by asserting ARREADY.

Following a handshake where both ARVALID and ARREADY are de-asserted, the Slave provides the requested data on the Read Data channel and asserts RVALID to denote its validity, thus completing the transaction.

Conversely, in a write transaction, the Master asserts AWVALID for the address and WVALID for the data, alongside BREADY to signal readiness for a response. The Slave acknowledges by asserting AWREADY and WREADY for the address and data, respectively. Handshakes on both Write Address and Write Data channels precede the Slave's assertion of BVALID, indicating a valid response. The transaction reaches completion on the subsequent clock edge. These meticulously orchestrated steps ensure seamless communication between the Master and Slave, preserving data integrity and facilitating transaction finalization.

**iv. Drawing Algorithm of VRAM and Font ROM**

- The algorithm for drawing text characters from the VRAM and font ROM involves accessing the font ROM module with an 11-bit address, where the row number serves as the address input, and the entire row of data (an 8-bit chunk) is outputted. Each symbol in the font sprite table occupies 16 rows in the ROM. The formula for locating a symbol in the font sprite table is determined by its starting and ending addresses, where the starting address is calculated as 16 * n and the end address as (16 * n) + 15, with n corresponding to the hexadecimal code in the table. This process allows traversal of the font sprite table, providing the necessary data for rendering text characters on the display.
- Specifically, the addresses to index into the VRAM in Week 1 is (DrawX/32) + 20 * (DrawY/16), where DrawX and DrawY is a pair of coordinates VGA controller is drawing at. The index into the font ROM is co-determined by (DrawX, DrawY) and text character code stored in 7 bits of the 32-bit word. Since there’re 4 such text character codes in a 32-bit word, we need (DrawX%32) / 8 to determine, i.e. (DrawX%32) / 8 specifies which 7-bit text character code we are looking for.
- By zero-extending the 7-bit text character code to 11 bits, we then obtain the address into font ROM (n mentioned in the above paragraph). Now it’s time to consider how DrawX and DrawY make an influence on the indices. Choosing which one within the range of 16 * n and (16 * n) + 15 is determined by DrawY, i.e. we are looking for 16 * n + (DrawY%16). Similarly, choosing which bit along fontROM[16 * n + (DrawY%16)] is determined by DrawX. i.e. we are looking for fontROM[16 * n + (DrawY%16)][(DrawX%32) % 8].

**v. Implementation of the Inverse Color Bit and the Control Register**

In the implementation, the inverse color bit (INVn) is utilized to indicate whether a character should be drawn with inverted colors. When INVn is set to 1, it signifies that the character should be drawn with inverted colors, meaning '1's are drawn with the background color, and '0's are drawn with the foreground color. Conversely, when INVn is set to 0, the character is drawn with regular colors, where '1's are drawn with the foreground color and '0's are drawn with the background color.

| Bit | 31-25 | 24-21 | 20-17 | 16-13 | 12-9 | 8-5 | 4-1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Function | UNUSED | FGD_R | FGD_G | FGD_B | BKG_R | BKG_G | BKG_B | UNUSED |

The control register contains information about foreground and background colors for all text characters, i.e. there would only be exactly one foreground color and exactly one background color on the screen. Foreground color will be stored in bits 24-13, containing 4-bit R(ed), 4-bit G(reen) and 4-bit B(lue). Background color will be stored in bits 12-1, containing 4-bit R(ed), 4-bit G(reen) and 4-bit B(lue). The other bits are unused in control register. Each time we draw an character, we will acquire the content in the control register (word address 0x258) and decomposes foreground color and background color. Combined information from the inverse bit of the text character, we can correctly draw every text character.

**b. Week 2 (Color Text Display)**

**i. Description of the Hardware Changes**

**1. Modification of Register-based VRAM to On-chip Memory-based VRAM**

- Since there’re only a single read port and a single write port to on-chip memory, we need to calculate the index to VRAM using DrawX and DrawY beforehand to acquire the correct 32-bit word, and this index occupies the read port. As for the write port, there’re two situations. One is writing 32-bit text character word, and the other one is writing 32-bit palette color word.
- Specifically speaking on the usage of the write port, in the modified module ram_32x8, an array mem of 32-bit words is declared to represent the on-chip memory. During write operations, if the 12th bit of the write address is 0, indicating a write to the on-chip memory, data is written to the memory location specified by the write address. Similarly, if the 12th bit of the write address is 1, indicating a write to the palette memory, data is written to the corresponding location in the palette array. Read operations are performed similarly, with data being read from the specified memory location. This modification enables the use of on-chip memory for storing video data, enhancing efficiency and performance compared to register-based VRAM.

2. **Corresponding Modifications to the IP Editor**

   To support multi-color text character drawing, we need to substitute video driver code (software driver) from supporting single-color text drawing to files supporting multi-color text drawing, involving a .c and a .h file.

3. **Modified Sprite Drawing Algorithm**

   - The addresses to index into the VRAM in Week 2 is (DrawX/16) + 40 * (DrawY/16), where DrawX and DrawY is a pair of coordinates VGA controller is drawing at. The index into the font ROM is co-determined by (DrawX, DrawY) and text character code stored in 7 bits of the 32-bit word. Since there're 2 such text character codes in a 32-bit word, we need (DrawX%16) / 8 to determine, i.e. (DrawX%16) / 8 specifies which 7-bit text character code we are looking for.
   - By zero-extending the 7-bit text character code to 11 bits, we then obtain the address into font ROM (n mentioned in the above paragraph). Now it's time to consider how DrawX and DrawY make an influence on the indices. Choosing which one within the range of 16 * n and (16 * n) + 15 is determined by DrawY, i.e. we are looking for 16 * n + (DrawY%16). Similarly, choosing which bit along fontROM[16 * n + (DrawY%16)] is determined by DrawX. i.e. we are looking for fontROM[16 * n + (DrawY%16)][(DrawX%16) % 8].

4. **Additional Modifications to Support Multicolored Text**

   When storing 32-bit words through AXI bus, we need to determine whether it is a text character word or palette colors information. If the word address is within the range of 0x000 and 0x4AF, it is a text character word, which should be stored using on-chip memory. If the word address is within the range of 0x800 and 0x807, it is a palette color word, which should be stored using AXI registers. Other word addresses are reserved but unused.

5. **Additional Hardware/code to Draw Palette Colors**

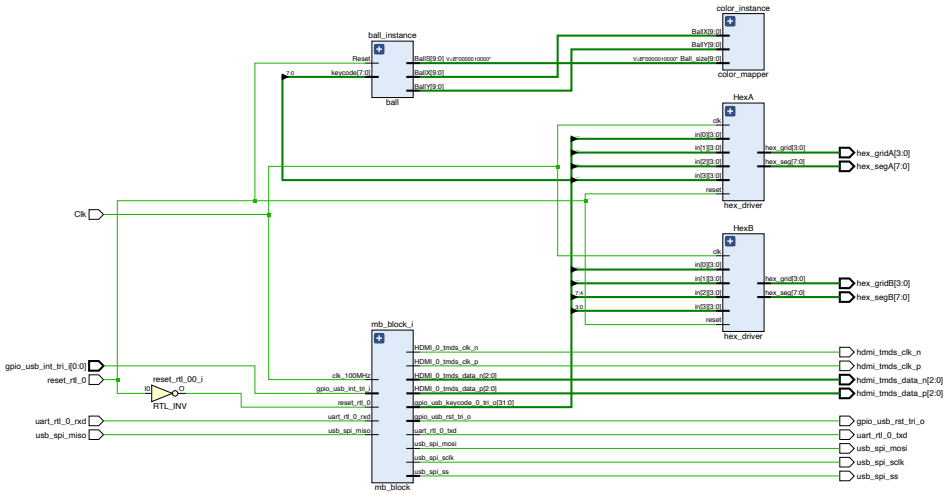
   We need to determine which foreground and background color we are going to use from the color index stored in every 32-bit text character word. If the signal "char_on" is high, indicating no inverse in drawing current character, the RGB values are assigned from specific bit ranges of a register array indexed by the foreground color index into the palette registers. Otherwise, if "char_on" is low, indicating an inverse in drawing current character, the RGB values are assigned based on conditions related to the position of another entity indexed by foreground color index into the palette registers. Additionally, we need to determine whether the indices are odd or even numbers to indexing into the palette registers since there're two colors stored

in a 32-bit word in the palette registers.

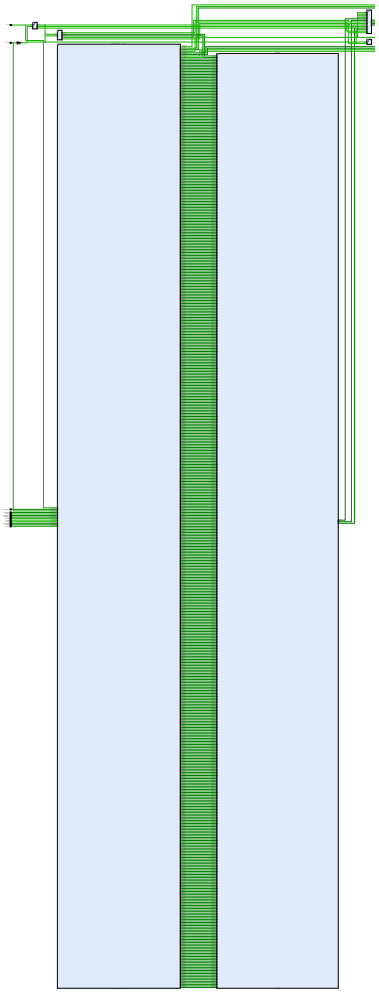
## 3. Block Diagram

### a. Week 1 (Monochrome Text Display)

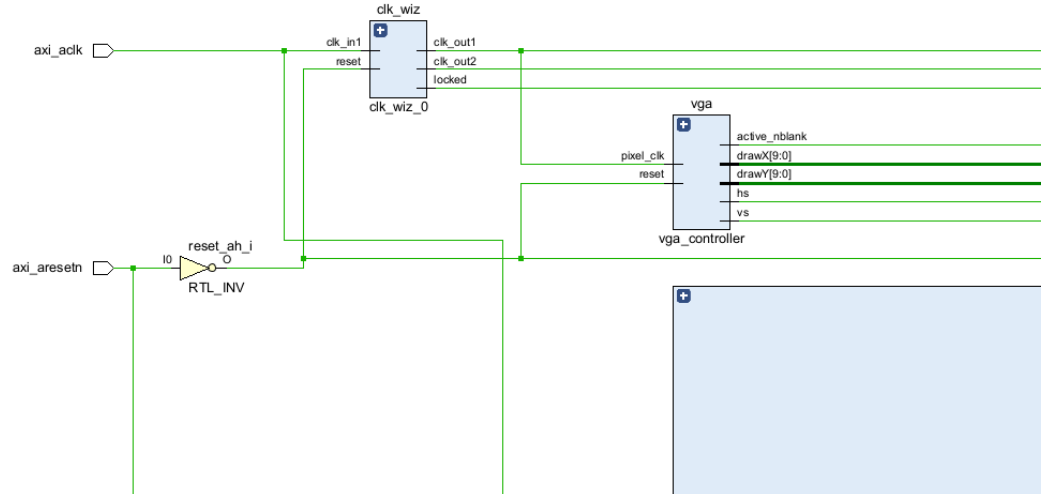
#### ➢ Top Level



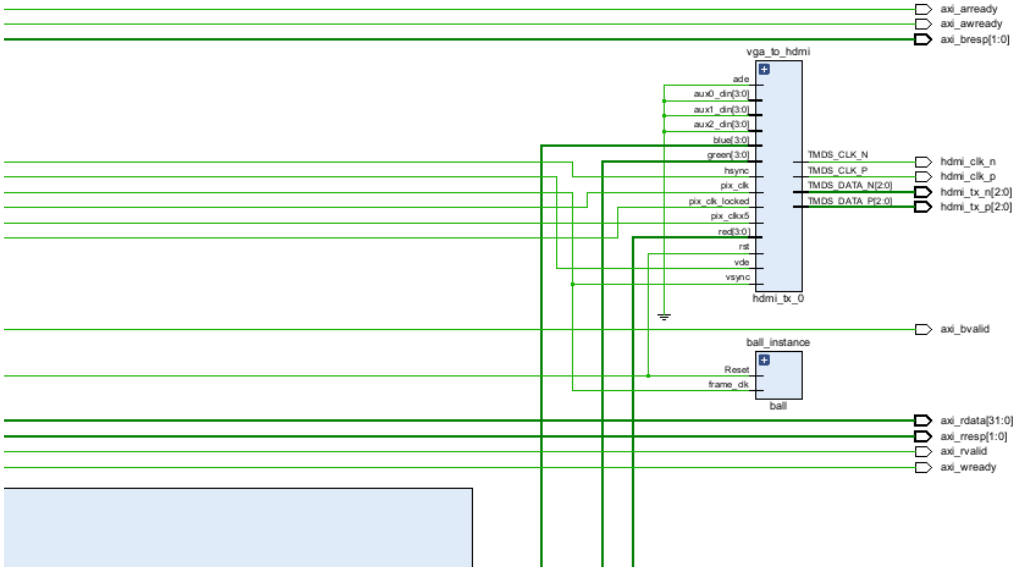

#### ➢ Internal Block Diagram of the IP



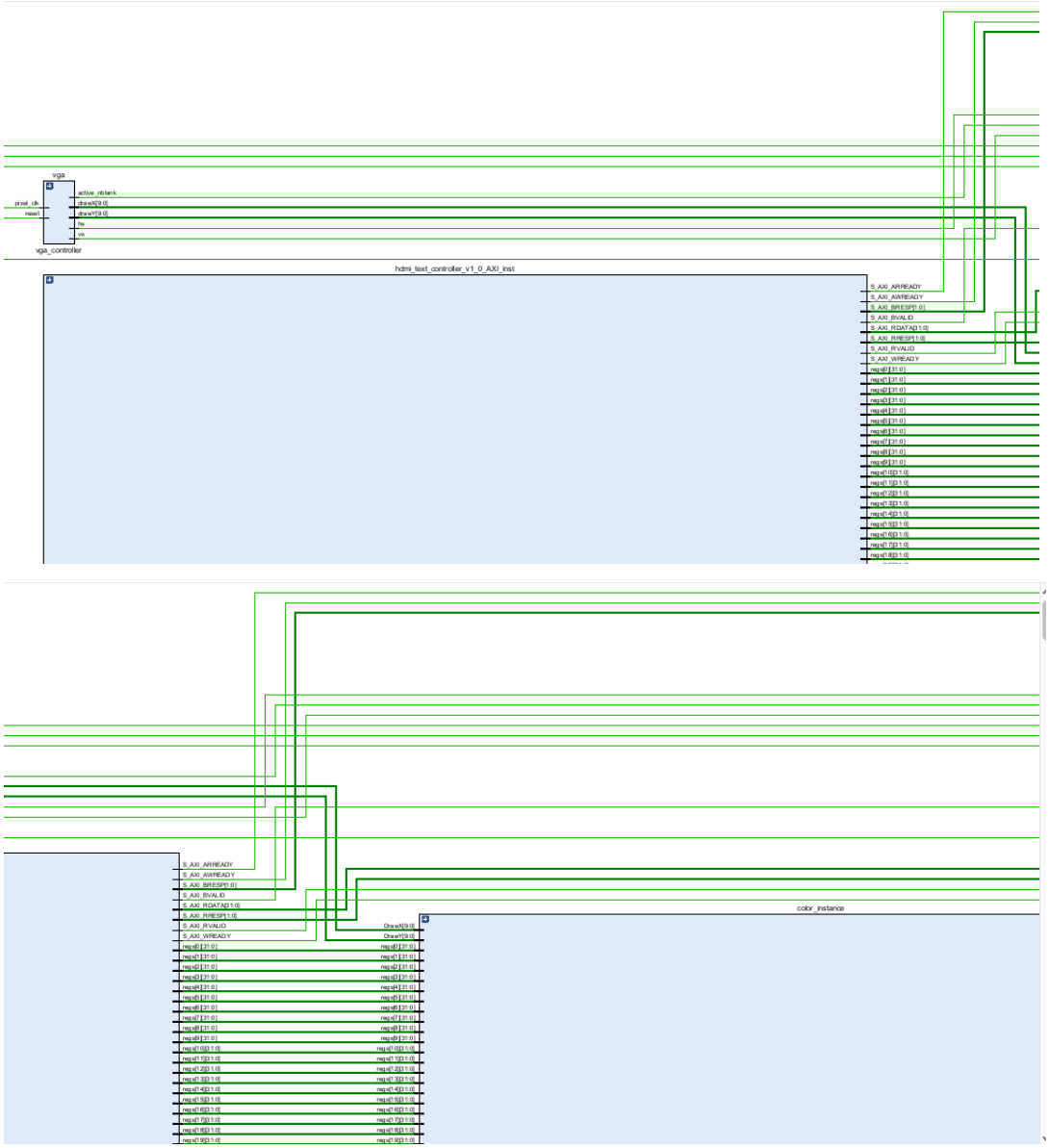
## Upper left corner:



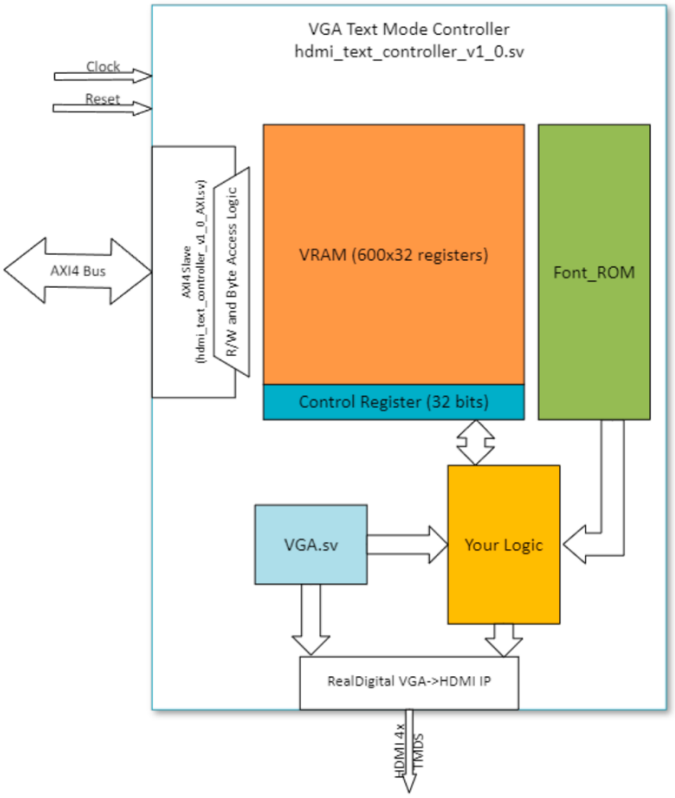

## Upper right corner:



**In the middle (The color mapper module and the AXI bus transmits a 32-bit register of 601 words)**



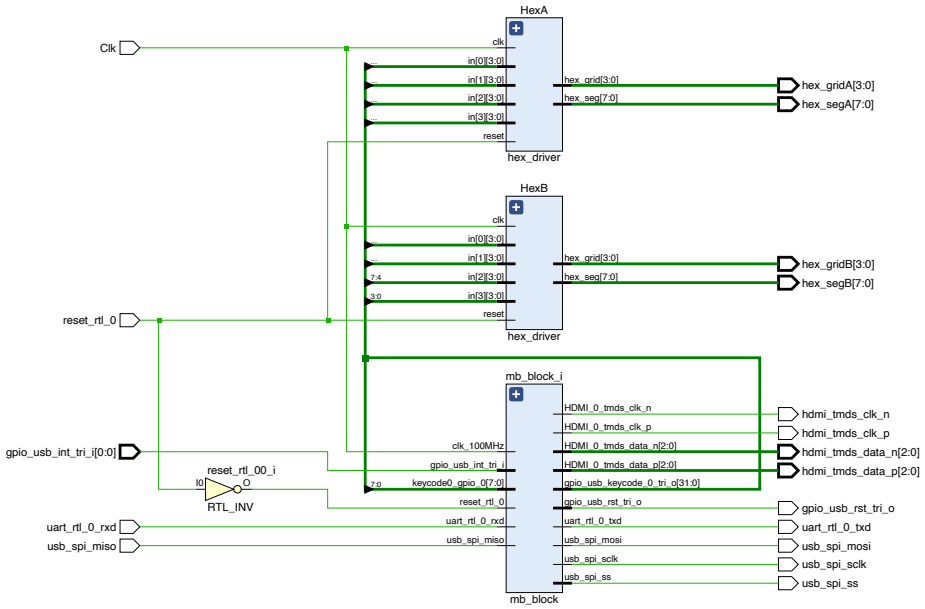
**Big View Block Diagram:**



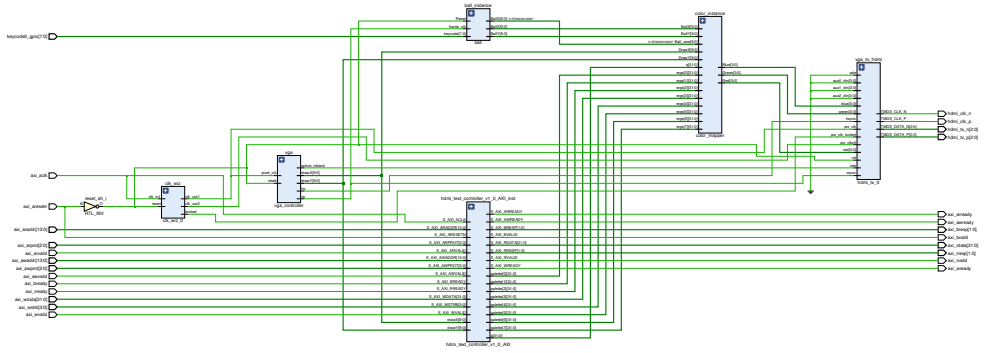

"Your Logic" is the middle part where the color mapper module and the AXI bus transmits the content of a 32-bit register containing 601 words.

**b. Week 2 (Color Text Display)**

- **Top Level**



➢ **Internal Block Diagram of the IP**



## 4. Module Descriptions

### a. Shared Components of Week 1 and Week 2

➢ **Top Level Modules (Including the Block Design Files)**

*Module:* mb_usb_hdmi_top.sv (Lab 7.1, modified on Lab 6.2)

*Inputs:* Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd

*Outputs:* gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB

*Description:* This file contains a top-level module designed for integrating various peripherals including USB, UART, HDMI, and VGA. It uses an integrated IP to generate HDMI signals. It can also serve as a bridge between ball motion and USB keycode control. This module interfaces with multiple blocks including USB, UART, HDMI text controller, hex displays, a ball module, and a color mapper.

*Purpose:* Integrating USB, UART, HDMI, VGA, and additional functionalities within a single top-level file (module).

*Module:* hex_driver.sv

*Inputs:* clk, reset, [3:0] in[4]

*Outputs:* [7:0] hex_seg, [7:0] hex_grid

*Description:* This file contains a module of visualizing outputs. To be specific, it contains a module of demonstrating output values on LEDs using 7-segment method. An LED digit on the Urbana Board is divided into 7 segments, with different segment combinations we can express numbers from 0 to F in hexadecimal.

*Purpose:* To translate hex numbers (from 0 to F) into figures appearing on LEDs.

*mb_block Module:*

This module is likely the MicroBlaze block, responsible for implementing the MicroBlaze processor in the FPGA design.

Interface signals:
clk_100MHz: Clock signal at 100 MHz provided to the MicroBlaze block.
gpio_usb_int_tri_i: Tri-state input signal for USB interrupt from the USB controller.
gpio_usb_keycode_0_tri_o, gpio_usb_keycode_1_tri_o: Tri-state output signals for keyboard scan codes received from the USB controller.
gpio_usb_rst_tri_o: Tri-state output signal for USB reset control.
reset_rtl_0: Active low reset signal for the MicroBlaze block.
uart_rtl_0_rxd, uart_rtl_0_txd: UART receive and transmit signals.
usb_spi_miso, usb_spi_mosi, usb_spi_sclk, usb_spi_ss: SPI interface signals for communication with the USB controller.

*clk_wiz_0 Module:*
This module is likely a clock wizard or clock generator block, responsible for generating clock signals with specific frequencies required by the HDMI interface.
Interface signals:
clk_out1: Output clock at 25 MHz, used for VGA timing.
clk_out2: Output clock at 125 MHz, used for HDMI timing.
reset: Active high reset signal for the clock wizard.
locked: Output signal indicating whether the generated clocks are locked and stable.
clk_in1: Input clock signal provided as the reference clock input to the clock wizard.

*AXI GPIO:*
It is a fundamental interface commonly found in microcontrollers and FPGAs, allowing them to interact with the external world by controlling and monitoring digital signals. In this context, GPIO signals are often used to interface with various peripherals, such as buttons, switches, LEDs, and other devices.

*AXI UART:*
It is a commonly used serial communication interface found in microcontrollers, FPGAs, and other electronic devices. It enables asynchronous serial communication between devices, allowing them to exchange data over serial links.

*AXI TIMER:*
The AXI Timer module represents a programmable timer component designed to generate periodic or one-shot timing events in FPGA designs. AXI Timer modules

are commonly used for tasks such as scheduling tasks, measuring time intervals, and triggering events at specific time intervals.

*AXI Quad SPI:*

The AXI Quad SPI module represents the Advanced eXtensible Interface (AXI) Quad Serial Peripheral Interface (SPI) controller, commonly used in FPGA designs to interface with SPI flash memories, sensors, and other SPI-compatible devices. The AXI Quad SPI controller provides a high-speed serial communication interface between the FPGA and external SPI devices, supporting quad data lines for increased data transfer rates.

*AXI INTC:*

The AXI INTC serves as the central hub for managing interrupts within the MicroBlaze system, facilitating communication between the processor and peripheral devices. It handles interrupt requests generated by various peripherals connected to the MicroBlaze processor, prioritizing and dispatching them to the processor for handling.

*AXI Peripheral Interface:*

This block serves as the bridge between the MicroBlaze processor and the AXI interconnect, facilitating communication between the processor and various peripheral devices in the system. The AXI Peripheral Interface block typically supports multiple AXI interfaces, allowing it to connect to multiple peripherals simultaneously.

- **Internal Modules of the IP**

  *clk_wiz_0 Module:*

  This module is likely a clock wizard or clock generator block, responsible for generating clock signals with specific frequencies required by the HDMI interface.

  Interface signals:

  clk_out1: Output clock at 25 MHz, used for VGA timing.

  clk_out2: Output clock at 125 MHz, used for HDMI timing.

  reset: Active high reset signal for the clock wizard.

  locked: Output signal indicating whether the generated clocks are locked and stable.

  clk_in1: Input clock signal provided as the reference clock input to the clock wizard.

  *hdmi_tx_0 Module:*

This module is likely the HDMI transmitter block, responsible for converting VGA signals to HDMI-compatible signals for display on an HDMI monitor.
Interface signals:
Various clocking and reset signals for synchronization.
Color and synchronization signals (red, green, blue, hsync, vsync, vde) for HDMI output.
Differential outputs (TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N) for HDMI TMDS signaling.

*Module:* VGA_controller.sv (Lab 6.2)
*Inputs:* pixel_clk, reset
*Outputs:* hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
*Description:* This file contains a module of generating VGA video signals, which requires synchronizing the horizontal and vertical timing of the display as well as managing the drawing coordinates within the screen resolution. It finally outputs several necessary signals for activating the display pixels including x and y pixel coordinates, horizontal and vertical drawing frequencies, etc. It contains internal counters for horizontal and vertical positions to provide proper timing for horizontal and vertical sync pulses. Besides, it also takes reset conditions and display range into consideration.
*Purpose:* Generating VGA related signals and pixel coordinates for further drawing as well as displaying on the screen after converting to HDMI signals.

*Module:* font_rom
*Inputs:* [10:0] addr
*Outputs:* [7:0] data
*Description:* This module serves as a read-only memory (ROM) containing font data. It has a parameterized width for address and data, defined by ADDR_WIDTH and DATA_WIDTH respectively. The ROM holds font data for characters encoded in 8-bit patterns. Each character occupies a block of 8 bits within the ROM, allowing for quick access based on the address input. The font data represents characters in a 16x8 pixel matrix.
*Purpose:* To provide font data for display purposes, enabling the generation of characters on a screen or display device based on input addresses.

*Module:* hdmi_text_controller_v1_0_AXI
*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

*Outputs:* S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA, S_AXI_RRESP, S_AXI_RVALID, regs

*Description:* This module includes logic for handling AXI4-Lite transactions, memory mapped register selects and write, as well as read operations. The module interfaces with a global clock signal, reset signal, write, and read address/data channels, and various control signals for AXI transactions.

*Purpose:* To provide an AXI4-Lite interface for accessing registers and handling transactions.

*hdmi_tx_0 Module:*

This module is likely the HDMI transmitter block, responsible for converting VGA signals to HDMI-compatible signals for display on an HDMI monitor.

Interface signals:

Various clocking and reset signals for synchronization.

Color and synchronization signals (red, green, blue, hsync, vsync, vde) for HDMI output.

Differential outputs (TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N) for HDMI TMDS signaling.

**b. Week 1 Unique Components**

*Module:* Color_Mapper.sv

*Inputs:* [9:0] DrawX, DrawY, [31:0] regs[601]

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

*Description:* This module processes drawing coordinates and register data to determine the color to display at each pixel on the screen. It incorporates font data and a circular boundary check to differentiate between the ball's area and the background, assigning the appropriate RGB values accordingly.

*Purpose:* To map drawing coordinates to RGB values for display on a screen, considering the position of the ball and the background area.

*Module:* hdmi_text_controller_v1_0

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

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

*Description:* This module serves as an HDMI AXI4 IP controller for generating text on a screen. It includes parameters for data and address widths and incorporates clocking, VGA sync signal generation, VGA to HDMI conversion, and ball movement

functionalities. The color mapper within the module determines the RGB values to display at each pixel based on the drawing coordinates.
*Purpose:* To control HDMI output and text rendering on a screen, mapping drawing coordinates to RGB colors for display.

**c. Week 2 Unique Components**

*Module:* Color_mapper.sv
*Inputs:* [9:0] DrawX, DrawY, [31:0] regs[8], [31:0] q
*Outputs:* [3:0] Red, Green, Blue
*Description:* This module computes the RGB color values to be displayed at each pixel on the screen based on the relative position of the ball and the current drawing coordinates. It incorporates font data to determine if a pixel falls within the ball's area or the background area, subsequently assigning appropriate RGB values. Additionally, it utilizes register data to determine the color values and implements logic for color selection based on ball presence and position.
*Purpose:* To map locations (positions) to RGB colors for display on the screen.

*Module:* ram_32x8
*Inputs:* [31:0] d, [11:0] write_address, read_address, we, clk, S_AXI_ARESETN, slv_reg_wren, C_S_AXI_DATA_WIDTH, register, [3:0] S_AXI_WSTRB
*Outputs:* [31:0] q, data_out, [31:0] palette[8]
*Description:* This module implements a 32x8 RAM with dual-port access to complete on-chip memory design. It stores data in a memory array "mem" and a palette array "palette" based on write and read addresses. The module handles write and read operations, considering byte-level enable signals and reset conditions, and outputs the appropriate data through signals "q" and "data_out". The palette is utilized for addresses exceeding 2048 using registers.
*Purpose:* To provide memory storage and retrieval functionality with dual-port access for data and palette values in a 32x8 RAM configuration to complete on-chip memory design.

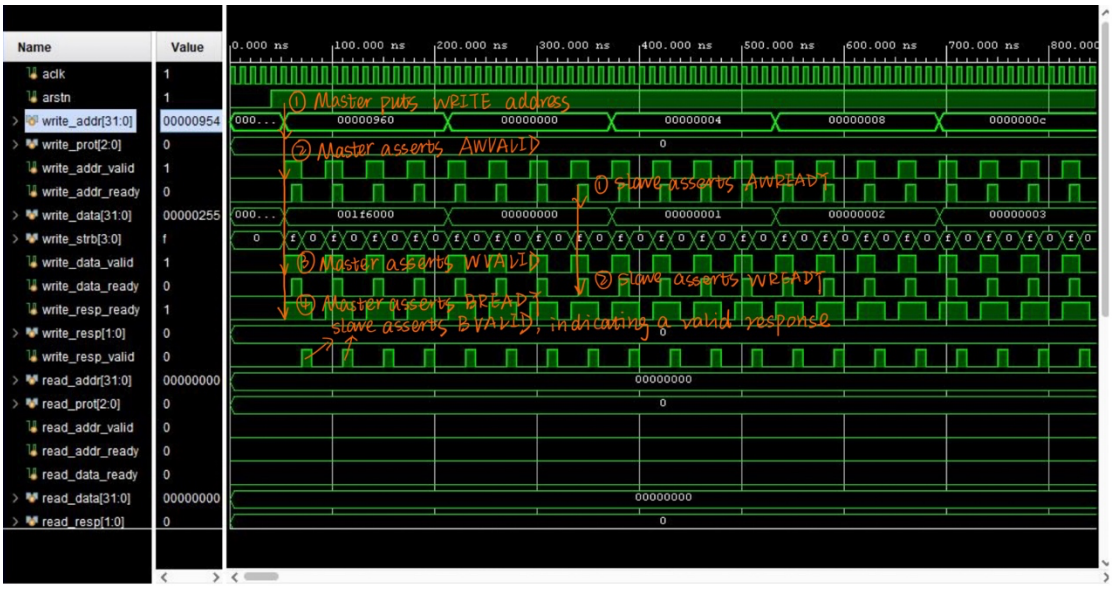
*Module:* hdmi_text_controller_v2_0
*Inputs:* [31:0] d, [11:0] write_address, read_address, we, clk, S_AXI_ARESETN, slv_reg_wren, C_S_AXI_DATA_WIDTH, register, [3:0] S_AXI_WSTRB
*Outputs:* [31:0] q, data_out, [31:0] palette[8]
*Description:* This module serves as a controller for an HDMI text display system, interfacing with an AXI bus for data communication. It includes logic for memory storage and retrieval, as well as for handling write and read operations. Additionally, it instantiates clocking and synchronization modules for VGA and HDMI signals, along

with a ball module for graphical representation.

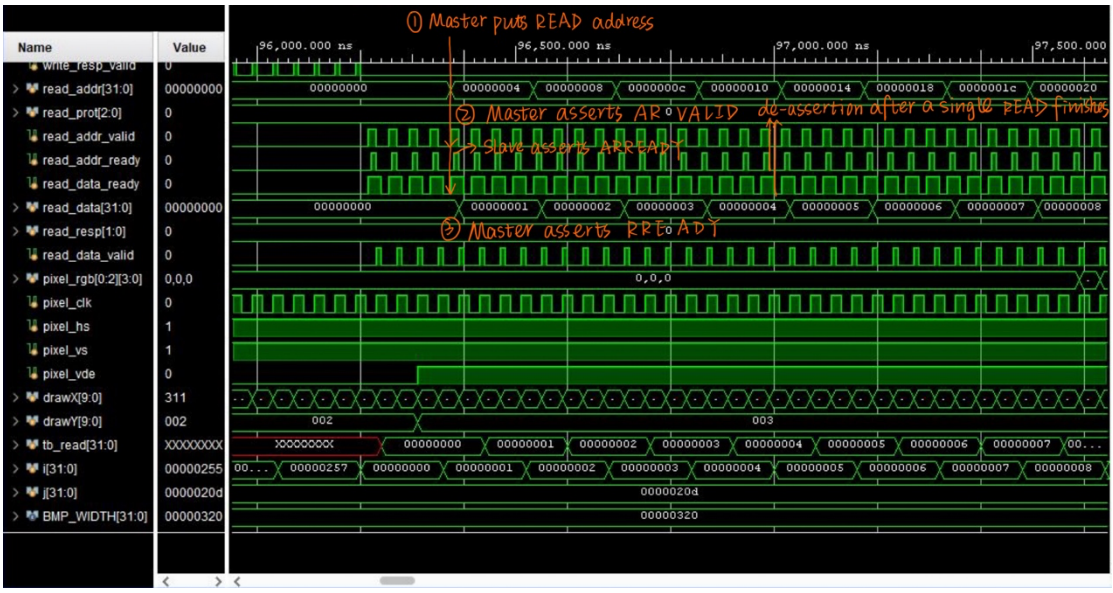
*Purpose:* To manage HDMI text display functionality, facilitate communication with an AXI bus, and coordinate clocking and synchronization signals for VGA and HDMI displays, incorporating graphical elements for enhanced visual representation.

## 5. Simulation Waveforms and Images

### a. AXI Write Transaction (Using Week 1's design)




### b. AXI Read Transaction (Using Week 1's design)




### c. Detailed Description / Commented Code of the axi_read ()

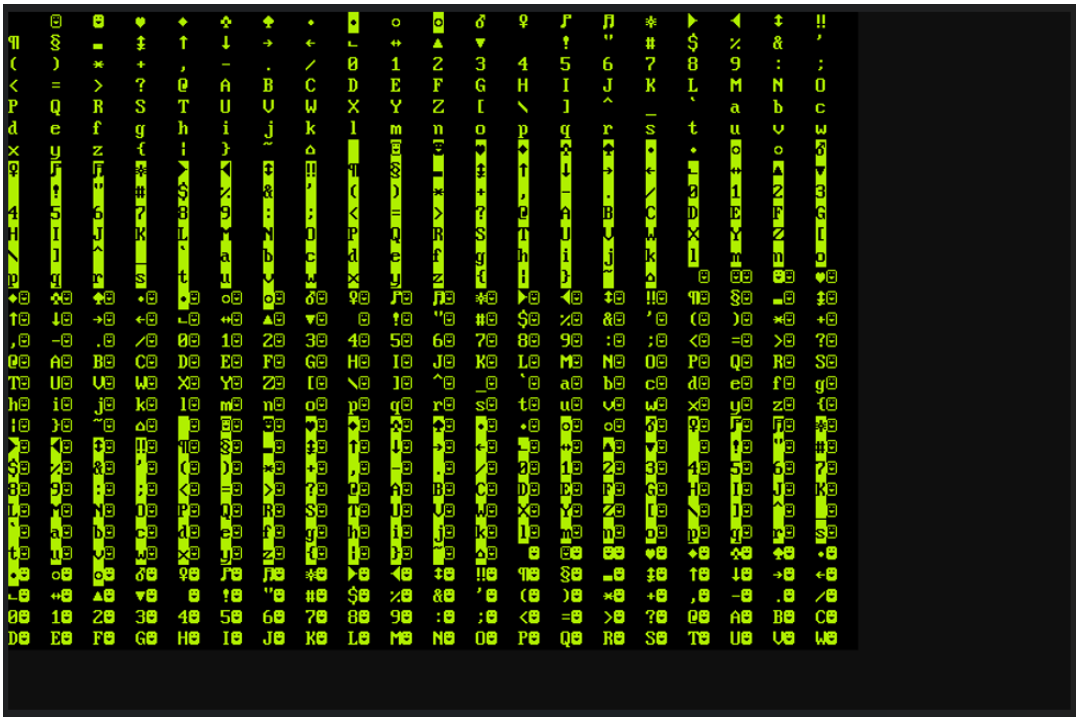
- Upon invocation of the task, it first waits for a delay of 3 time units (#3) before assigning the input address addr to the read_addr signal. It then asserts the read_addr_valid (ARVALID) and read_data_ready (RREADY) signals to indicate that the address is valid and the data is ready to be read.
- The task then waits until the read_addr_ready (ARREADY) signal is asserted, indicating that the receiver is ready to accept the address.

- Once the read_addr_ready signal is detected, the task samples the read_data signal and assigns its value to the data output. It then waits for the read_data_valid signal to be asserted, indicating that valid data is available.
- Upon detection of the read_data_valid signal, the task waits for a positive edge of the clock signal (aclk). Upon this edge, it deasserts the read_addr_valid and read_data_valid signals, completing the read operation handshake.

```
task axi_read (input logic [31:0] addr, output logic [31:0] data);
    begin
        #3 read_addr <= addr;               // waits for a delay of 3 time units, then assigning the input address addr
        read_addr_valid <= 1'b1;            // asserts the read_addr_valid (ARVALID)
        read_data_ready <= 1'b1;            // asserts the read_data_ready (RREADY)

        wait(read_addr_ready);              // waits for slave asserting ARREADY

        @(posedge aclk);
        begin
            read_addr_valid<=1'b0;          // de-asserts the read_addr_valid (ARVALID)
            read_addr_ready<=1'b0;          // de-asserts the read_data_ready (RREADY)
        end

        data <= read_data;                  // read data

        wait(read_data_valid);

        @(posedge aclk);
        begin
            read_data_valid<=1'b0;
            read_data_ready<=1'b0;
        end
    end
endtask;
```

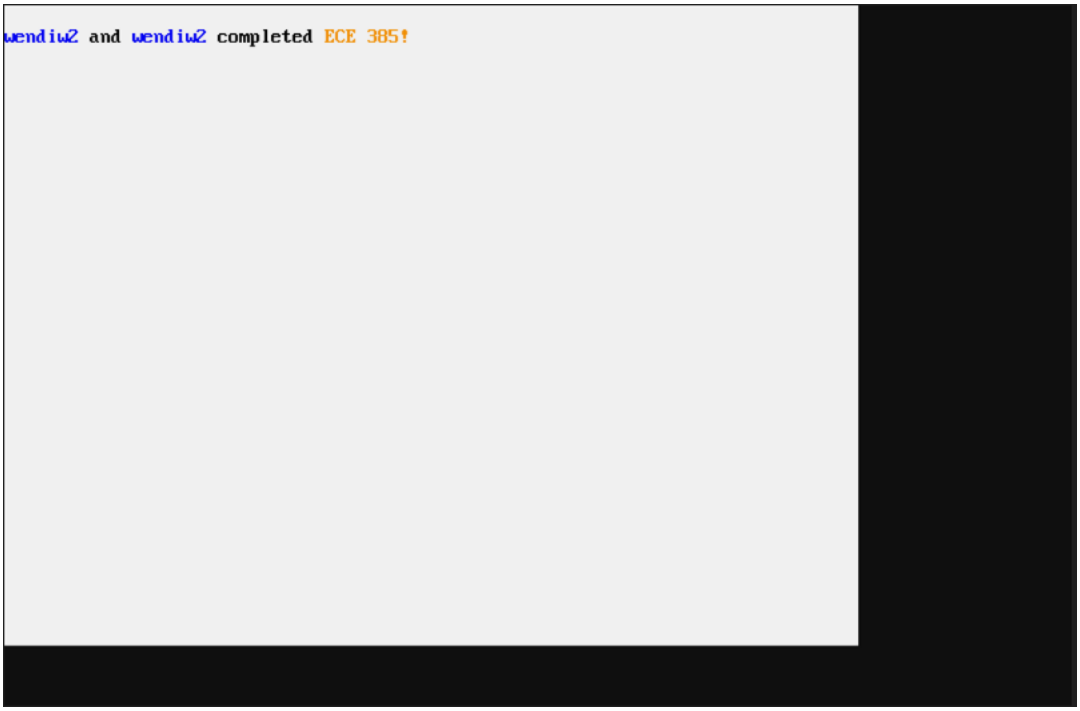
- 

**d. Simulated Image from Week 1 Testbench**



**e. Simulated Image from Week 2**

Printing the string in the second line with chosen white background:




**f. The Commented Portion of The Modified Test Bench**

The commented portion is about writing text characters and colors into memory using axi_write function. We still use some for loops to write contents into memory through axi_write function. In 5.e., we need to write the string “wendiw2 and wendiw2 completed ECE 385!”, where “wendiw2” is written in blue (F, 0, 0) and “ECE 385!” is written in orange (F, 9, 0).

Before modification:

```
//remember AXI addresses are BYTE addresses!
//This writes something into the Control Register so that we're not simulating a black screen
repeat (4) @(posedge aclk) axi_write((600*4), 32'h001F6000); //write control reg to set foreground and background

//Write into every one of the 600 VRAM registers, note that this is different than what the driver C code does
//because the testbench axi_write task only generates aligned (full 32-bit) AXI writes (e.g. write_strb is always F)
//The C code on the MicroBlaze expects to be able to do byte and halfword (16-bit) writes, therefore if the
//simulation works but the checksum does not pass in the hardware, check handling of write_strb.
for(i=0; i < 600; i++) begin
  repeat (4) @(posedge aclk) axi_write(4*i, i);
end
```

After modification:

```
323         for(i=0; i < 40; i++) begin
324           repeat (4) @(posedge aclk) axi_write(4*i, 32'h20222022);  // White background, all spaces
325         end
326         repeat (4) @(posedge aclk) axi_write(4*40, 32'h65027702);   // we
327         repeat (4) @(posedge aclk) axi_write(4*41, 32'h64026e02);   // nd
328         repeat (4) @(posedge aclk) axi_write(4*42, 32'h77026902);   // iw
329         repeat (4) @(posedge aclk) axi_write(4*43, 32'h20023202);   // 2 + space
330         repeat (4) @(posedge aclk) axi_write(4*44, 32'h6e326132);   // an
331         repeat (4) @(posedge aclk) axi_write(4*45, 32'h20326432);   // d + space
332         repeat (4) @(posedge aclk) axi_write(4*46, 32'h65027702);   // we
333         repeat (4) @(posedge aclk) axi_write(4*47, 32'h64026e02);   // nd
334         repeat (4) @(posedge aclk) axi_write(4*48, 32'h77026902);   // iw
335         repeat (4) @(posedge aclk) axi_write(4*49, 32'h20023202);   // 2 + space
336         repeat (4) @(posedge aclk) axi_write(4*50, 32'h6f326332);   // co
337         repeat (4) @(posedge aclk) axi_write(4*51, 32'h70326d32);   // mp
338         repeat (4) @(posedge aclk) axi_write(4*52, 32'h65326c32);   // le
339         repeat (4) @(posedge aclk) axi_write(4*53, 32'h65327432);   // te
340         repeat (4) @(posedge aclk) axi_write(4*54, 32'h20326432);   // d + space
341         repeat (4) @(posedge aclk) axi_write(4*55, 32'h43124512);   // EC
342         repeat (4) @(posedge aclk) axi_write(4*56, 32'h20124512);   // E + space
343         repeat (4) @(posedge aclk) axi_write(4*57, 32'h38123312);   // 38
344         repeat (4) @(posedge aclk) axi_write(4*58, 32'h21123512);   // 5!
345         for(i=59; i < 1200; i++) begin
346           repeat (4) @(posedge aclk) axi_write(4*i, 32'h20222022);  // White background, all spaces
347         end
348
349         repeat (4) @(posedge aclk) axi_write(4*2048, 32'h0013FE00); // Index 0: Blue (0, 0, F), Index 1: Orange (F, 9, 0)
350         repeat (4) @(posedge aclk) axi_write(4*2049, 32'h00001FFE); // Index 2: White (F, F, F), Index 3: Black (0, 0, 0)
```

## 6. Documentation of Design Sources and Statistics

| | Week 1 | Week 2 |
|---|---|---|
| LUT | 16253 | 8009 |
| DSP | 3 | 4 |
| Memory (BRAM) | 32 | 35 |
| Flip-Flop | 22249 | 3227 |
| Frequency | 77.18 MHz | 99.80 MHz |
| Static Power | 0.077 W | 0.077 W |
| Dynamic Power | 0.423 W | 0.394 W |
| Total Power | 0.5 W | 0.471 W |

### a. The Difference between Using On-chip Memory for VRAM and Registers

On the one hand, on-chip memory offers a buffer for efficient access and manipulation of graphical content, typically with larger capacity and specialized features to support real-time rendering of high-resolution graphics.

On the other hand, registers are small, fast-access memory elements used for temporary storage and control within processors or peripheral devices. They store intermediate results, status flags, and control signals, providing quick access to critical data for processing and control operations.

In summary, on-chip memory has larger space to store bits and compiles faster, but it takes longer to access memory stored in it. However, registers are fast to access while costing more time in compiling.

### b. Efficiency Discussion and Tradeoffs

- Week 2's design (using on-chip memory) is more efficient since it has a higher frequency. The tradeoff in using on-chip memory for VRAM and registers lies in the balance between access speed and storage capacity. While on-chip memory

offers larger capacity and specialized features for real-time rendering of high-resolution graphics, it typically incurs longer access times compared to registers. Registers, on the other hand, provide fast access to critical data but have limited capacity. In the context of Week 2's design, which favors on-chip memory for its higher frequency, the tradeoff involves sacrificing some access speed for the benefits of increased storage capacity and specialized features. Therefore, the efficiency gained in terms of higher frequency comes at the expense of slightly longer access times, highlighting the inherent tradeoff between speed and capacity in memory utilization.

- Additionally, in Week 1's design, we need to transmit all 32-bit words (601 32-bit words) stored in registers between the color mapper module and the AXI bus when we perform reading from memory. However, in Week 2's design, we only need to transmit one 32-bit word when reading from on-chip memory, thus the reading efficiency is largely enhanced.

## 7. Conclusion

### a. Functionality of This Design

- **Week 1 Functionality**

  All required functionalities work well in our design. This design implements a text mode graphics controller connected to an AXI4-Lite memory-mapped bus and supporting 80 column text mode through HDMI output. The functionality includes rendering text characters onto the display by accessing font data from a provided font ROM asynchronously. Each character is represented as an 8-bit code, allowing for a total of 128 characters from codepage 437. The VRAM holds the pixel data for the characters, with each 32-bit word accommodating four characters. The implementation supports the option of drawing characters with inverted colors, indicated by an inverse color bit. Additionally, a control register is incorporated to offer further customization options for drawing text, such as setting color palettes and adjusting text attributes. Overall, this design provides a simple yet efficient solution for displaying text on a screen, suitable for various applications requiring text-based output.

- **Week 2 Functionality**

  All required functionalities work well in our design. This design extends Week 1's text mode graphics controller from single color to supporting multiple colors. The functionality includes rendering text characters onto the display by accessing font data from a provided font ROM asynchronously. Each character is represented as a 16-bit code, allowing for a total of 128 characters from codepage 437. The VRAM holds the pixel data for the characters, with each 32-bit word accommodating two characters. The implementation supports the option of drawing characters with

inverted colors, indicated by an inverse color bit. Additionally, instead of using the control register, this week's design uses palette registers containing 16 colors in eight 32-bit words. With palette indices contained in each pixel data, each text character can have its own foreground and background colors. Since each text character occupies 16-bit information, Week 2's design is also extended from using AXI registers to on-chip memory, in support of 1200 32-bit words information in total of drawing on a 640x480 screen.

**b. Potential Extensions**

- In Week 2's design, we learned how to store contents in on-chip memory, which extends the space of storing bits. It will be very helpful for the Final Project if we want to design an animation, requiring a high amount of storage.
- In the software part of Week 2's design, there is a way of self-extending the screen contents and colors if we can combine the randomness of setting colors in C code and the control of moving objects in hardware design. If a self-extending / self-updating algorithm can be implemented, it would be very helpful in a game designing Final Project.

**c. Summary**

I think Week 1's instructions are kind of ambiguous, and we are confused what we need to do in Week 1. We finally find some useful and instructional information in the comments of the provided files. Maybe there could be a list of what to do in the Lab 7 manual.