ECE 385

Spring 2024

Lab 7

Ziheng Li (zihengl5)

Xin Yang (xiny9)

Section AL1

Prof. Zuofu Chen

Introduction

HDMI interface

HDMI was used to transfer video+audio data from computer to monitor. In the context of this lab, the HDMI interface was used to display text onto a screen. The HDMI interface just connects the color_mapper module and reads the synchronization data along with color information corresponding to a specific pixel. The ultimate objective is to understand the connection between memory-mapped interface (AXI4) and the HDMI output, which allows us to draw characters on a monitor.

Modifications from Lab 6.2

The design in Lab 7 builds upon the basic hardware and software framework in Lab 6.2. In Lab 6.2, the focus was on creating a System-On-Chip design that interacted with peripherals like LEDs, switches, and VGA. To achieve that, we create several input/output ports inside block design. This builds the foundation components inside the FPGA environment. In Lab 7.1, we remove the input/output ports. But instead, we introduce the HDMI IP, text mode controller, as well as off-chip memory (registers). This change requires a more sophisticated control over the memory mapping (using the AXI4 bus) and handling of graphic data. In lab 7.2, we extended our design in 7.1 with on-chip memory and more color controls. This allows us to better utilize the given source on FPGA.

Lab 7&6.2 Hardware->Software Communication

In Lab 6.2, hardware-software communication has been accomplished through direct memory access. This method provides a straightforward and immediate way to communicate between the hardware and software, where we could treat hardware ports directly as memory. However, in Lab 7, the IP approach was used, which offers a more modular way for hardware-software communication. This approach encapsulates functionality into IP blocks, which can then be integrated and reused across different designs.

Advantages of IP approach:

1. The largest advantage is that IP cores allow us to reuse the file in different systems, enhancing design flexibility.
2. The second advantage is that IP cores can be developed independently. Designs can be easily scaled up or down by adding or removing IP cores.

Disadvantages of IP approach:

1. Within our implementation, we found it much harder to achieve hardware-software communication with IP approach. It requires specific read&write functions as well as a complicated hand-shaking process.

In general, IP approach in Lab 7 is more complex but is more structured and scalable compare to simple method used in Lab 6.2

Descriptions for Lab 7.1

i. Written Description of the Entire Lab 7 System:

The week 1 of Lab7 is designed to create a simplified text mode graphics controller that connects to the AXI4-Lite memory-mapped bus and outputs text mode graphics through an HDMI interface. The graphics controller supports an 80 column by 30 row text mode, for a total of 2400 characters. Each character is 16*8 pixels in size, and represented using 7 bits for glyphs with an additional bit to invert color. We stores all the text information in 600 32-bits register where each register could stores information for 4 characters. Color information was stored register #601.

ii. Description of the HDMI Text Mode Controller IP:

The HDMI Text Mode Controller IP is the core component responsible for generating the text mode graphics output to the HDMI interface. It interfaces with the AXI4-Lite bus to communicate with the MicroBlaze processor. This IP helps manage the registers to store the character data. It also implements the color mapper which draws the correct color of pixel to the screen based on the contents read from VRAM. Then it outputs the signal for HDMI video.

iii. AXI Register Read/Write Logic:

When reading the registers, the microBlaze initiates a read transaction and pushes the read address on the bus. Then the inside logic decodes the address to determine the number of registers which is going to be read. The next step is to assign the data to axi_rdata for output. The last step is to assert a ready signal to complete the transaction. When writing to the registers, the MicroBlaze processor initiates a write transaction by placing the write address on the AXI address bus, the write data on the AXI write data bus, and the write strobe to indicate which bytes of the write data are valid. Then it uses a 4 times loop to write to the specific byte of the registers one byte a time. The last step is to complete the write transaction when both the BREADY and BVALID are high. Overall speaking, read&write in Lab7.1 is similar accessing an array in C.

iv. Text Character Drawing Algorithm:

The algorithm used to draw the text characters on the HDMI output works as follows: The controller maintains a position (x, y) that tracks the current pixel being drawed. In the color mapper, we represent the coordinates of each pixel by DrawX and DrawY. We get the col number by dividing the DrawX by 32(right shift 5 bits), and get the row number by dividing the DrawY by 16(right shift 4 bits). Then the number of the registers is calculated by col + row*20. Then we have to determine the number of characters inside the register and corresponding pixel number in that row of characters. Character number: (DrawX % 32) / 8. Pixel number:(DrawX % 32) % 8. Since the memory content is little-endian, the first character's info is stored in the lower 8 bits of the register. And the address into the font is: character number * 16 + DrawY % 16. In our design we use the registers as an output from AXI file to color mapper so we did not calculate the VRAM address.

v. Inverse Color Bit and Control Register:

The inverse color bit in the most significant bit of the character information. Control register is the number 601 registers which stores two colors, font and back. We store the font RGB in [24:13] bits of the control register and the background RGB in [12:1] bits of the control register. When the inverse bit is high, we read the font RGB from [12:1], and background RGB from [24:13], and vice versa if the inverse bit is low.

Descriptions for Lab 7.2

i. VRAM to On-Chip memory:

In Lab 7.2, we need to transit register memory to on chip memory. In order to achieve that, we create an extra on-chip memory block inside our design. There are two ports on the memory block. On the AXI4 side, it connects to the C program in Vitis and it only requires writing to on chip memory, so we assign port "a" to AXI. Meantime, color mapper only needs to read from the on chip memory to get text and color information. Hence we assign another port "b" to color mapper. AXI and color_mapper are the only two parts that require on-chip-memory.

ii. Modifications in IP Editors:

In the IP Editor we modify the read and write method to align with the new VRAM requirements. Basically pass the address first and then read/write data, while in register, we could directly access contents synchronously, just like an array. Since the memory block has some latency for read&write, we also include a counter similar to Lab5 to wait for the contents after we give an address to it.

iii. Modified sprite drawing algorithm:

The modification for the sprite drawing algorithm is pretty simple. Since each word (32 bytes) is only 2 glyphs and it contains color information inside, we need to update the offset logic in order to locate the correct index of the character. First is the address finding part, since each character size increase to 16 bits, we only divide 16 instead of 32 to locate the x-address of current character. Second part is color, we need to locate the font and back color inside palette using bit 7:4 and bit 3:0 respectively of current character drawing.

iv. Text Character Drawing Algorithm:

To support multi-color drawing, each character now occupies 16 bits, where the first byte contains text information and the second byte contains indexes into the color palette. There are only two colors supported in Lab7.1, so we could set the two colors manually. However in Lab7.2, we need to dynamically go into the color palette and find the correct color for that text.

v. Inverse Color Bit and Control Register:

In order to support multiple color drawing, we introduce a color palette which stores RGB information. In our design, the color palette is a 16*16 register, and we could find the desired color by indexing into the color registers.

Block Diagram

Top level (Lab 7.1)

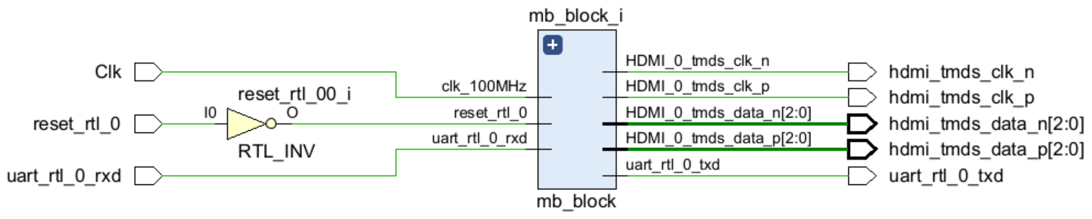


Fig.1 Overall view of top level (7.1)

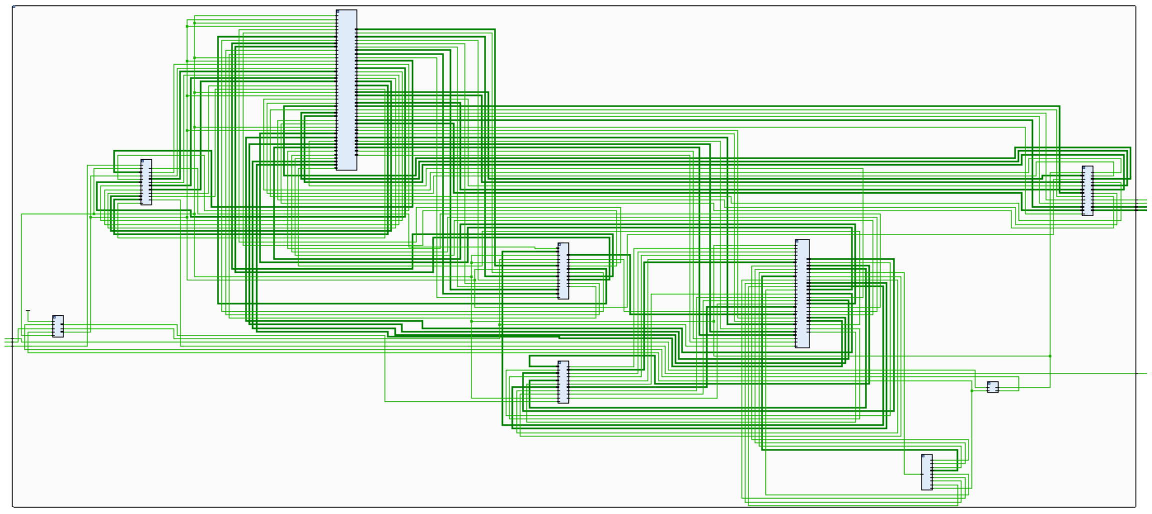


Fig.2 mb_block block diagram (7.1)

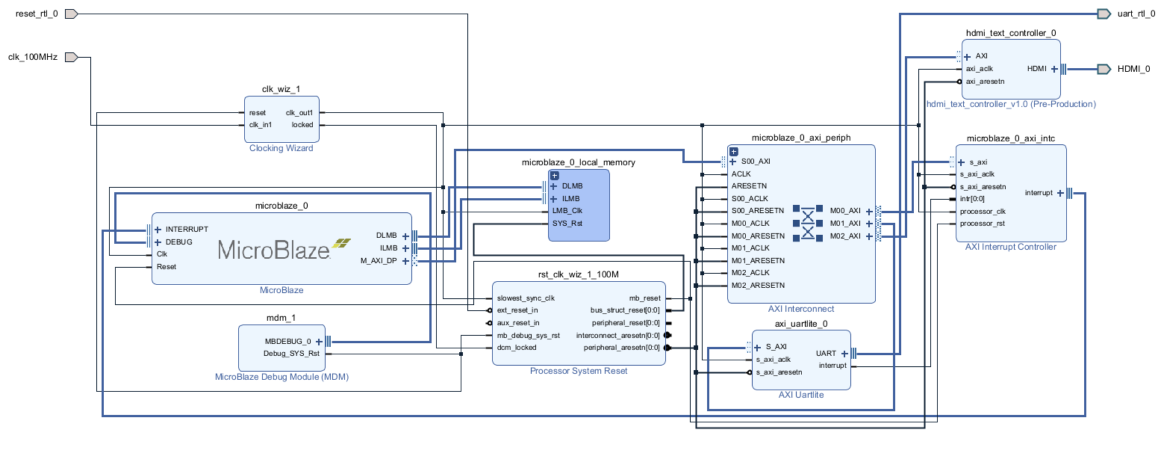


Fig.3 High level diagram (7.1)

Internal IP (Lab 7.1)

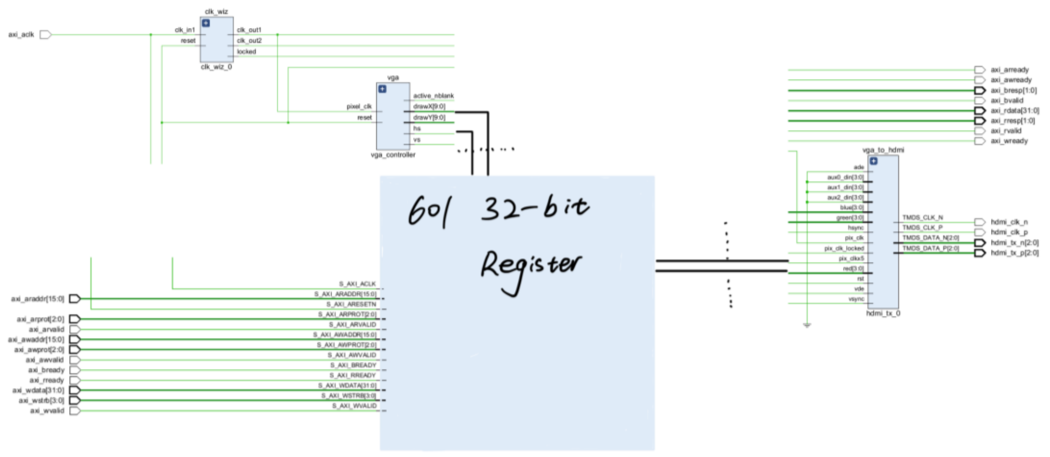


Fig.4 hdmi_to_text block diagram (7.1)

Top level (Lab 7.2)

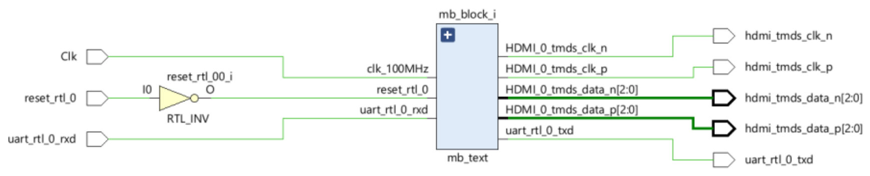


Fig.5 Overall view of top level (7.2)

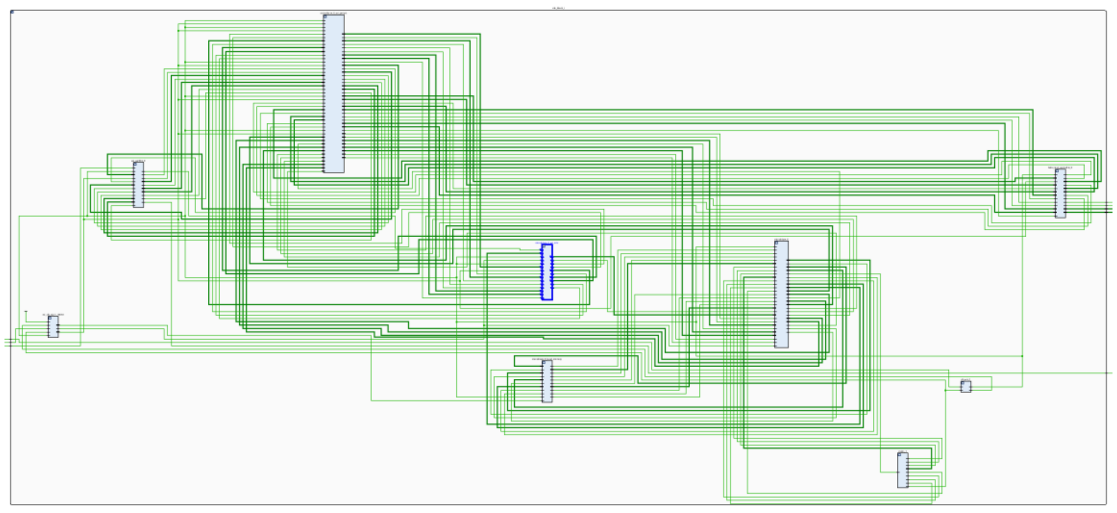


Fig.6 mb_text block diagram (7.2)

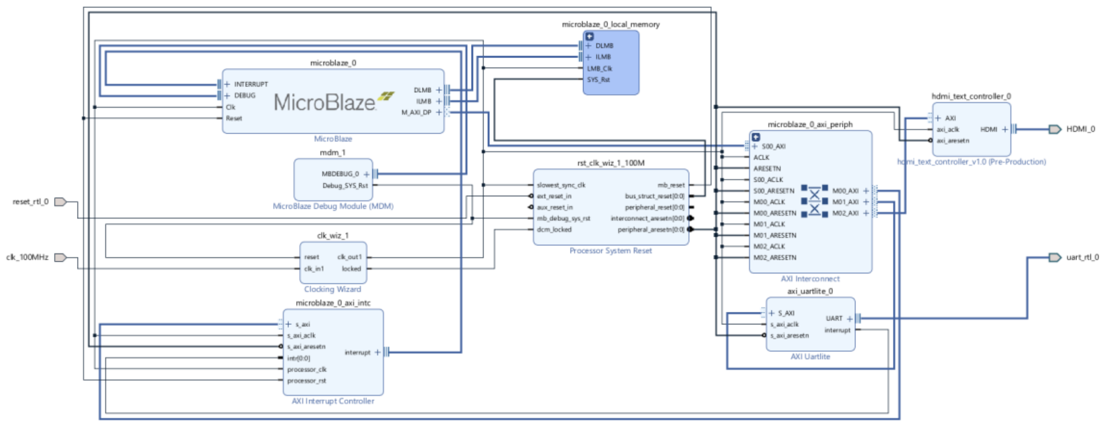


Fig.7 High level diagram (7.2)

Internal IP (Lab 7.2)

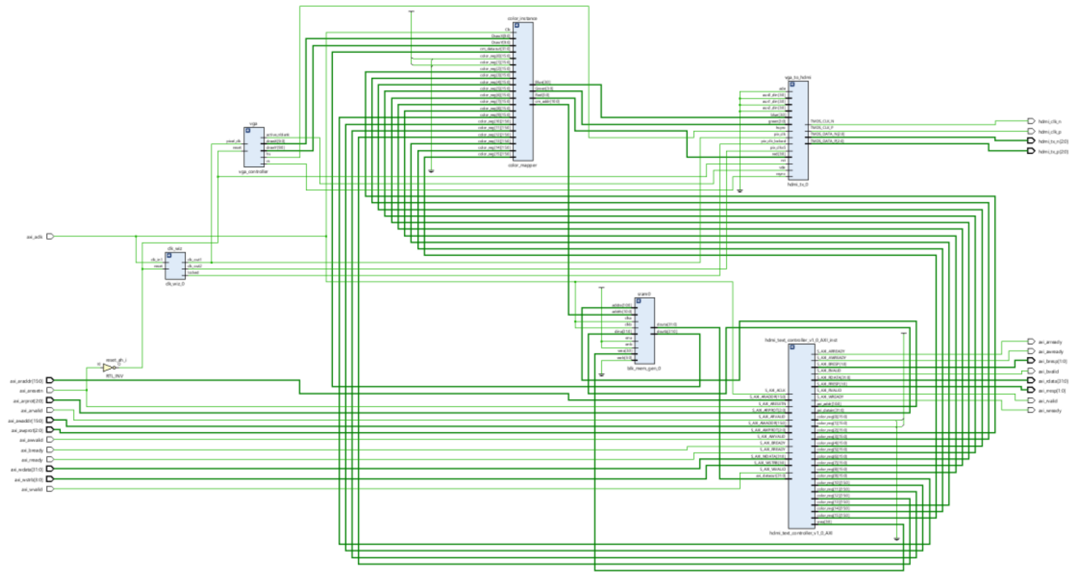


Fig.8 hdmi_to_text block diagram (7.2)

Module descriptions

Module: mb_usb_hdmi_top.sv
Inputs: Clk, reset_rtl_0, uart_rtl_0_rxd,
Outputs: uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0]hdmi_tmds_data_n, [2:0]hdmi_tmds_data_p
Description: This module instantiates the mb_block module, which is responsible for implementing the MicroBlaze system, UART, and HDMI functionalities.
Purpose: The purpose of this module is to provide a top-level interface for the USB and HDMI functionalities implemented in the mb_block module.

Module: mb_block.bd
Inputs: clk_100MHz, reset_rtl_0, uart_rtl_0_rxd
Outputs: uart_rtl_0_txd, HDMI_0_tmds_clk_n, HDMI_0_tmds_clk_p, [2:0]HDMI_0_tmds_data_n, [2:0]HDMI_0_tmds_data_p
Description: This module serves as the top-level block design for the MicroBlaze system. It instantiates and connects the necessary components to provide the HDMI and UART functionalities.
Purpose: The purpose of this module is to integrate the MicroBlaze processor, UART, and HDMI components into a complete system-on-chip.

Module: hdmi_text_controller_v1_0.sv
Inputs: axi_aclk, axi_aresetn, axi_awaddr[C_AXI_ADDR_WIDTH-1:0], axi_awprot[2:0], axi_awvalid, axi_wdata[C_AXI_DATA_WIDTH-1:0], axi_wstrb[C_AXI_DATA_WIDTH/8-1:0], axi_wvalid, axi_bready, axi_araddr[C_AXI_ADDR_WIDTH-1:0], axi_arprot[2:0], axi_arvalid, axi_rready
Outputs: axi_awready, axi_wready, axi_bresp[1:0], axi_bvalid, axi_arready, axi_rdata[C_AXI_DATA_WIDTH-1:0], axi_rresp[1:0], axi_rvalid, hdmi_clk_n, hdmi_clk_p, hdmi_tx_n[2:0], hdmi_tx_p[2:0]
Description: This module is the top-level design for an HDMI text controller IP block. It has 5 submodules that implement an AXI4 Slave interface for configuration and control, generates the necessary clocks, handles VGA synchronization, and includes a VGA to HDMI module. The module also contains a color mapper component that maps the text data to the appropriate color signals for display.
Purpose: The overall purpose of this module is to provide a HDMI text controller IP that can be easily integrated into a larger system-on-chip (SoC) design.

Module: hdmi_text_controller_v1_0_AXI.sv
Inputs: S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR[C_S_AXI_ADDR_WIDTH-1:0], S_AXI_AWPROT[2:0], S_AXI_AWVALID, S_AXI_WDATA[C_S_AXI_DATA_WIDTH-1:0], S_AXI_WSTRB[C_S_AXI_DATA_WIDTH/8-1:0], S_AXI_WVALID, S_AXI_BREADY, S_AXI_ARADDR[C_S_AXI_ADDR_WIDTH-1:0], S_AXI_ARPROT[2:0], S_AXI_ARVALID, S_AXI_RREADY
Outputs: S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP[1:0], S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA[C_S_AXI_DATA_WIDTH-1:0], S_AXI_RRESP[1:0], S_AXI_RVALID, mem_reg[31:0][601]
Description: This module is the AXI4-Lite interface for the HDMI text controller IP. It implements the necessary logic to handle the AXI4-Lite read and write transactions, including address decoding, data transfer, and response signaling. The module also provides access to an internal memory registers that can be used to store the text data to be displayed on the HDMI output.
Purpose: This module provides a standard AXI4-Lite interface for setting and operating the HDMI text controller IP. It wrapped the AXI4-Lite logic in a separate module which handles communication with the host CPU.

Module: clk_wiz_0.v
Inputs: reset, clk_in1
Outputs: clk_out1, clk_out2, locked
Description: This module is used to create a pixel clock (25MHz) and a 5x TMDS clock (125MHz).
Purpose: The purpose of this module is to provide 2 different clock signals.

Module: hdmi_tx_0.v
Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst, red, green, blue, hsync, vsync, vde, aux0_din, aux1_din, aux2_din, ade
Outputs: TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P[2:0], TMDS_DATA_N[2:0]
Description: The hdmi_tx_0 module is responsible for converting the input pixel data, synchronization signals, and auxiliary data into TMDS-encoded signals that can be transmitted over an HDMI link.
Purpose: The purpose of this module is to provide a ready-to-use HDMI transmitter solution that can be easily integrated into a larger FPGA-based design.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, frame, sync, [9:0] drawX, [9:0] drawY
Description: This module implements a VGA controller that generates the signals with proper frequency for displaying graphics on a VGA monitor. It generates the horizontal sync (hs) and vertical sync (vs) signals. It outputs the coordinates of the current pixel being drawn (drawX and drawY). Additionally, the module includes a frame signal that pulses once per frame in 60Hz, which can be used for the HDMI module to refresh the frames.
Purpose: The purpose of this module is to generate the timing signals and coordinate information required for displaying on a VGA monitor.

Module: color_mapper.sv
Inputs: DrawX[9:0], DrawY[9:0], mem_reg[601][31:0]
Outputs: Red[3:0], Green[3:0], Blue[3:0]
Description: This module is responsible for mapping the text data stored in the mem_reg array to the appropriate color signals for display on the HDMI output. It calculates the column and row indices into the mem_reg array based on the current pixel coordinates, extracts the current character to be displayed, looks up the corresponding font data in the font_rom module, and then maps the font bitmap data to the appropriate color signals based on the control register values, which determine the font and background colors.
Purpose: The purpose of this module is to provide the logic for converting the text data stored in the mem_reg array into the corresponding color signals that can be used to drive the HDMI output.

Module: font_rom.sv
Inputs: addr[10:0]
Outputs: data[7:0]
Description: This module defines a read-only memory (ROM) that stores the font bitmap data for the characters to be displayed on the HDMI output. The module takes an 11-bit address (addr) as input and outputs an 8-bit data value (data) that corresponds to the font bitmap for the requested character.
Purpose: The purpose of this module is to provide the font bitmap data that is used by the color_mapepr module.

Module: hdmi_text_controller_tb.sv
Inputs: NONE
Outputs: NONE
Description: This module serves as the testbench for the HDMI Text Controller IP. It verifies the correct functionality of the IP by simulating various AXI4-Lite transactions like read and write and generating a simulated image based on the IP's output signals.
Purpose: The purpose of this module is to test the HDMI Text Controller IP.

Week 2 major changes in modules

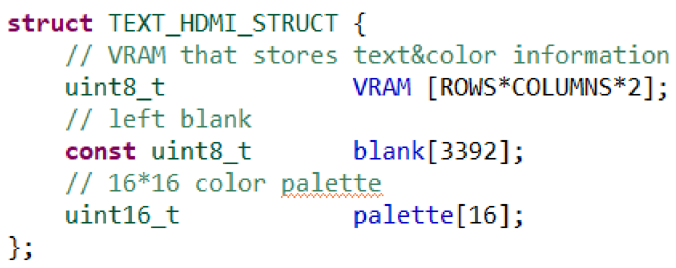
We added a block memory module blk_mem_gen_0 to replace the functionality of the mem_reg in week 1. In this case, implementation gets faster since we only left a few registers to store color information.

The most important changes are made within the AXI module. In this module we modify the write logic generation to distinguish when to write to color registers and video memory based on the 13th bit of address. And we assign the write strobe to wea when writing to video memory. Also we add a counter inside read and write logic to count for the latency of memory reading/writing.

Since each 32 bit address now only stores the font data and color data of 2 characters, corresponding changes are made inside the color mapper. The formula used to find the row is now divided by 4. Additionally, now we used a palette method to store up to 16 colors in the 16 registers. So now we assign the RGB based on the value stored in corresponding color registers instead of the control register.

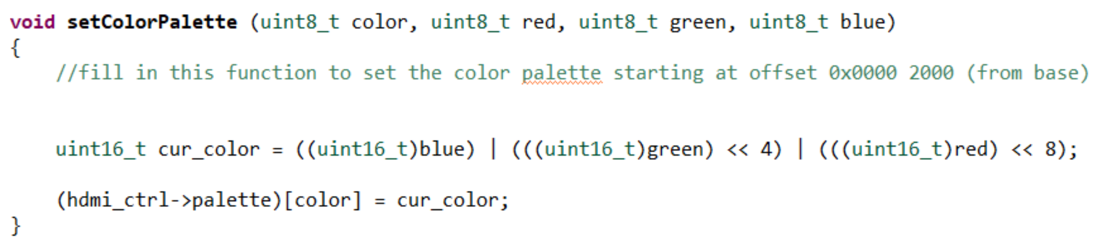
Modification in Vitis

In hdmi_text_controller.h, we implemented the struct design for TEXT_HDMI. We add 8*3392 blank space and 16*16 palette space.



```
struct TEXT_HDMI_STRUCT {
    // VRAM that stores text&color information
    uint8_t              VRAM [ROWS*COLUMNS*2];
    // left blank
    const uint8_t        blank[3392];
    // 16*16 color palette
    uint16_t             palette[16];
};
```

Fig.9 TEXT_HDMI_STRUCT



```
void setColorPalette (uint8_t color, uint8_t red, uint8_t green, uint8_t blue)
{
    //fill in this function to set the color palette starting at offset 0x0000 2000 (from base)


    uint16_t cur_color = ((uint16_t)blue) | (((uint16_t)green) << 4) | (((uint16_t)red) << 8);

    (hdmi_ctrl->palette)[color] = cur_color;
}
```

Fig.10 Set color function

Simulation of AXI Write

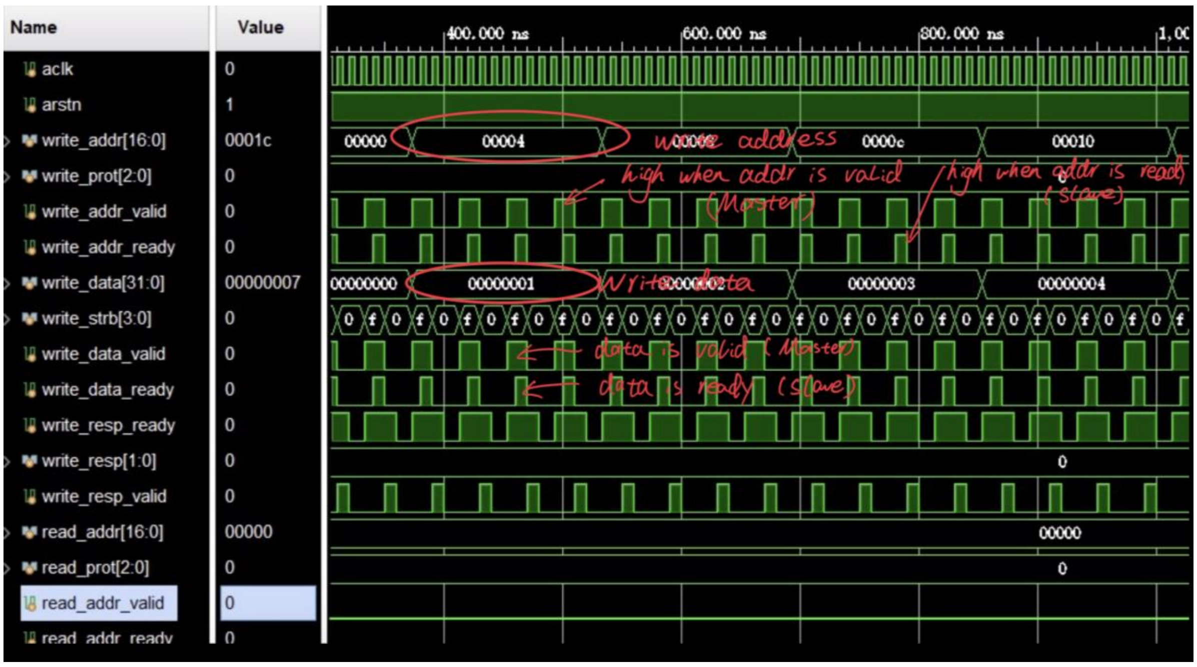


Fig.11 Simulation of AXI Write

Initialization: The master begins by placing an address on the Write Address channel and the corresponding data on the Write Data channel. It then asserts the AWVALID and WVALID signals to indicate that the address and data are valid and ready for transfer.

Acknowledgment: In response, the slave asserts the AWREADY and WREADY signals on the Write Address and Write Data channels, respectively. This indicates that the slave is ready to receive the address and data.

Handshake: With both valid and ready signals active on the Write Address and Write Data channels, handshakes occur on these channels, allowing the master to de-assert the AWVALID and WVALID signals as the slave gets the address and data.

Response & Completion: Subsequently, the slave asserts the BVALID signal, indicating that a valid response is ready on the Write Response channel. The write process will continue the next cycle.

Simulation of AXI Read

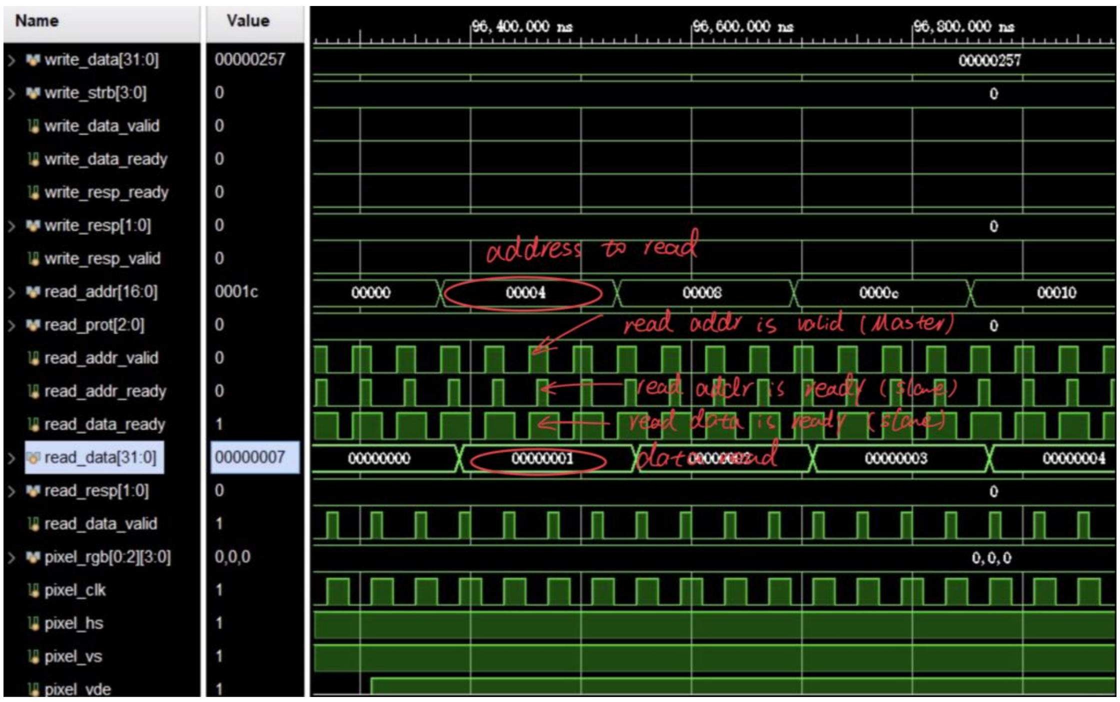


Fig.12 Simulation of AXI Read

Initialization: The master initiates the process by placing an address on the Read Address channel and asserting ARVALID to indicate that the address is valid.

Acknowledgment: The slave responds by asserting ARREADY, indicating it’s ready to accept the address from the master.

Handshake: With both ARVALID and ARREADY signals asserted, a handshake occurs at the next rising clock edge, after both the master and slave de-assert ARVALID and ARREADY. The slave now has the address.

Data Transfer & Completion: The slave places the requested data on the Read Data channel and asserts RVALID, indicating that the data is valid. The read process will continue the next cycle.

Simulation image from Week 1

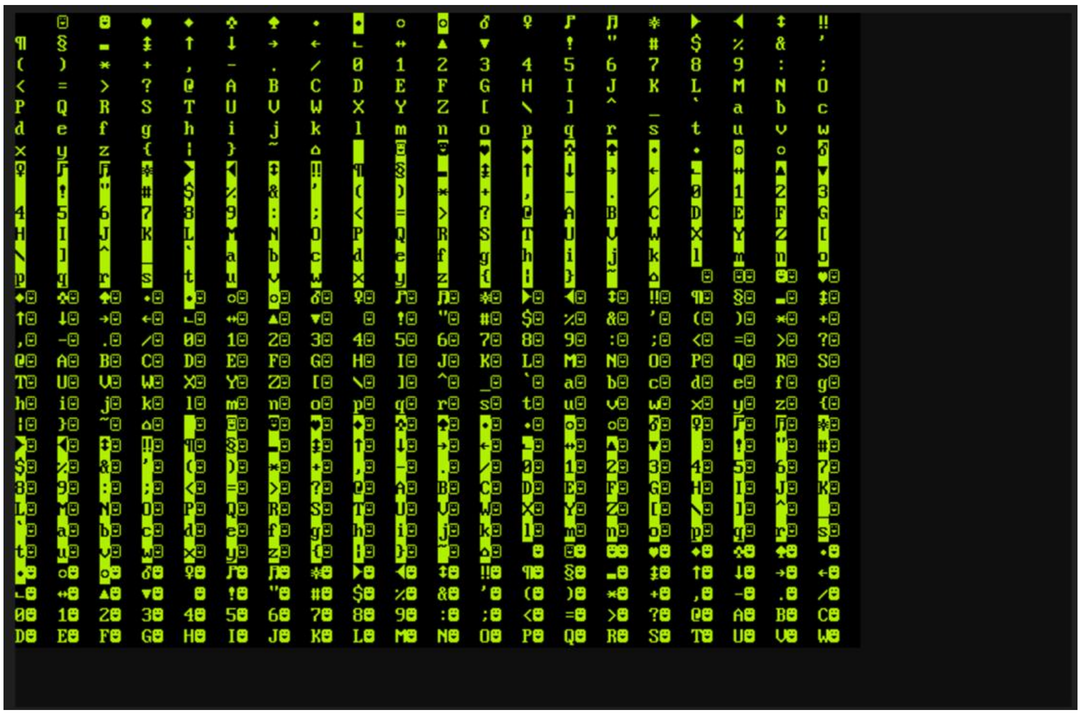
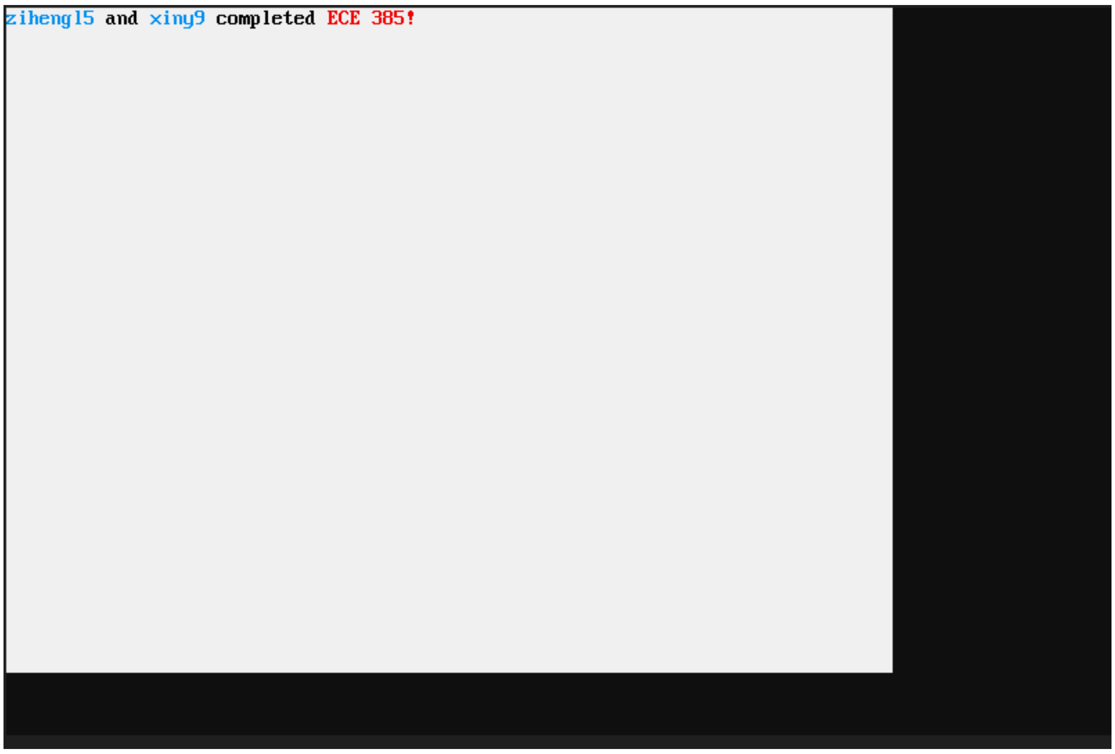


Fig.13 lab7_1_sim.bmp

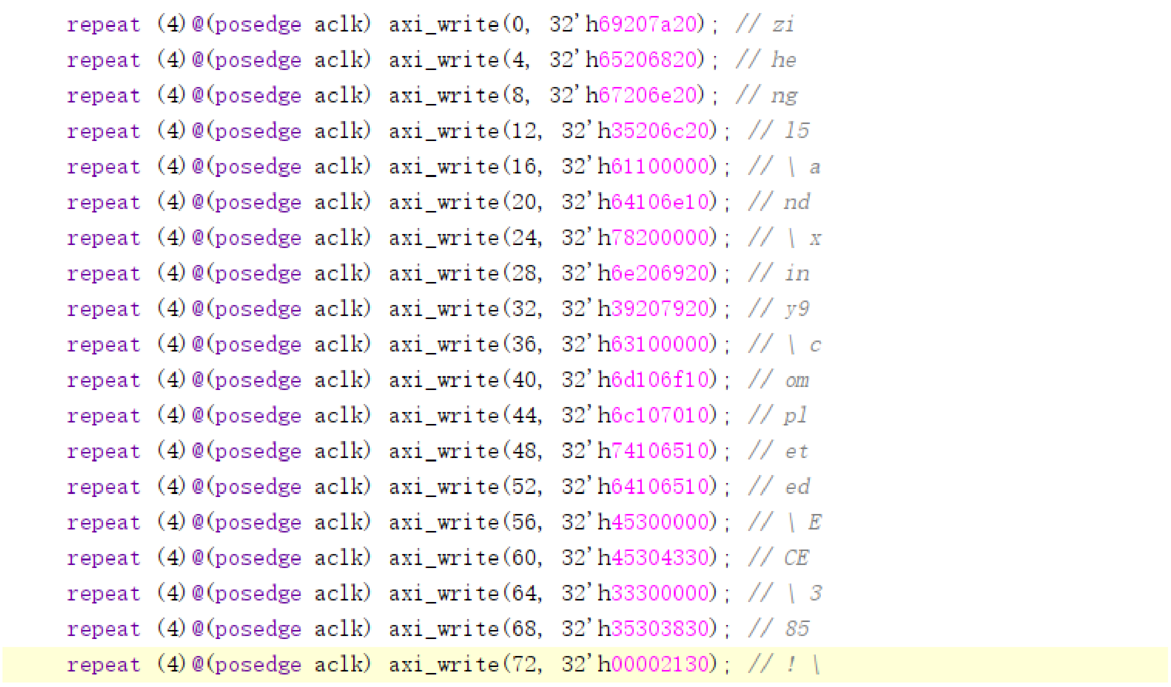
Simulation image from Week 2



ziheng15 and xing9 completed ECE 385!

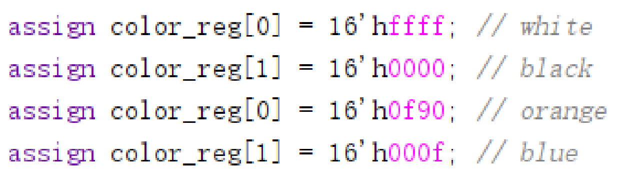
Fig.14 lab7_2_sim.bmp

Modified test bench and color register



```
repeat (4)@(posedge aclk) axi_write(0, 32'h69207a20); // zi
repeat (4)@(posedge aclk) axi_write(4, 32'h65206820); // he
repeat (4)@(posedge aclk) axi_write(8, 32'h67206e20); // ng
repeat (4)@(posedge aclk) axi_write(12, 32'h35206c20); // 15
repeat (4)@(posedge aclk) axi_write(16, 32'h61100000); // \ a
repeat (4)@(posedge aclk) axi_write(20, 32'h64106e10); // nd
repeat (4)@(posedge aclk) axi_write(24, 32'h78200000); // \ x
repeat (4)@(posedge aclk) axi_write(28, 32'h6e206920); // in
repeat (4)@(posedge aclk) axi_write(32, 32'h39207920); // y9
repeat (4)@(posedge aclk) axi_write(36, 32'h63100000); // \ c
repeat (4)@(posedge aclk) axi_write(40, 32'h6d106f10); // om
repeat (4)@(posedge aclk) axi_write(44, 32'h6c107010); // pl
repeat (4)@(posedge aclk) axi_write(48, 32'h74106510); // et
repeat (4)@(posedge aclk) axi_write(52, 32'h64106510); // ed
repeat (4)@(posedge aclk) axi_write(56, 32'h45300000); // \ E
repeat (4)@(posedge aclk) axi_write(60, 32'h45304330); // CE
repeat (4)@(posedge aclk) axi_write(64, 32'h33300000); // \ 3
repeat (4)@(posedge aclk) axi_write(68, 32'h35303830); // 85
repeat (4)@(posedge aclk) axi_write(72, 32'h00002130); // ! \
```

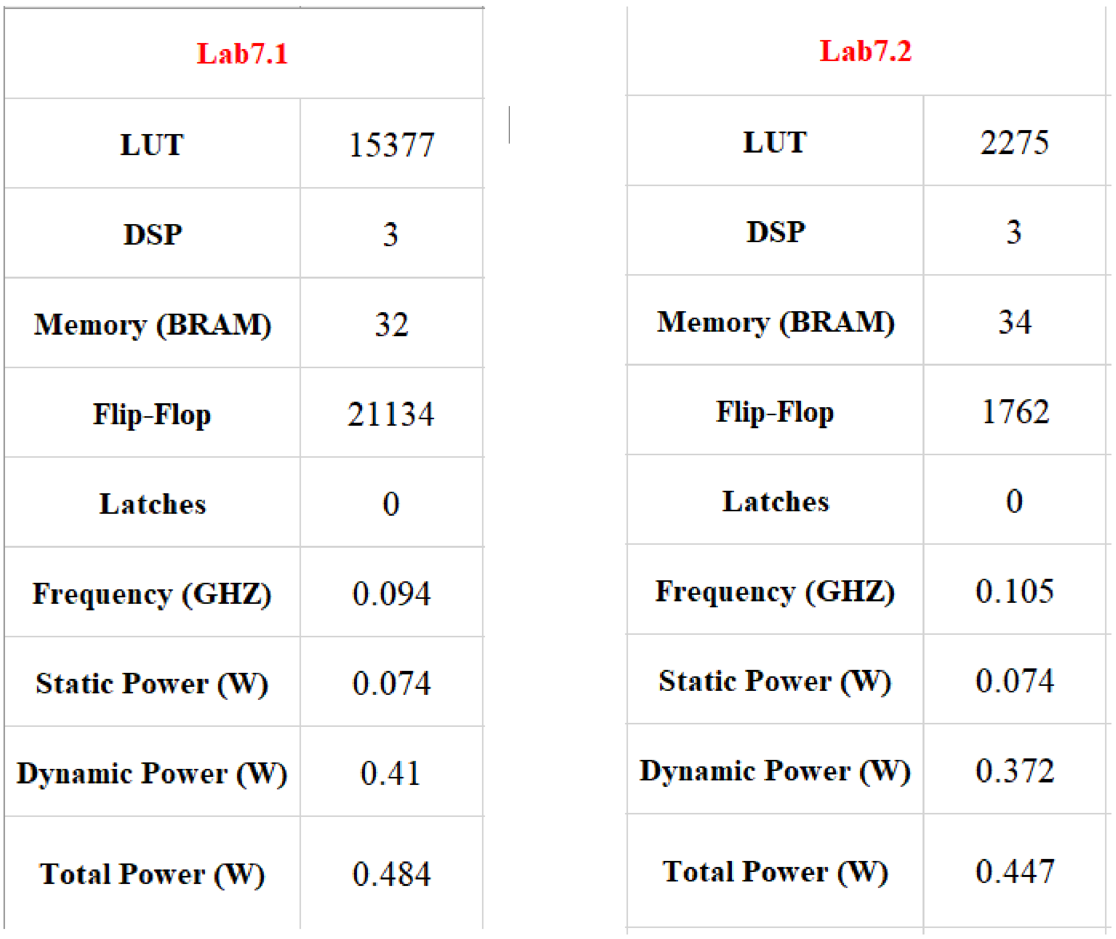
Fig.15 Modified test bench



```
assign color_reg[0] = 16'hffff; // white
assign color_reg[1] = 16'h0000; // black
assign color_reg[0] = 16'h0f90; // orange
assign color_reg[1] = 16'h000f; // blue
```

Fig.16 Modified color palette

Design Resources and Statistics



Lab7.1	
LUT	15377
DSP	3
Memory (BRAM)	32
Flip-Flop	21134
Latches	0
Frequency (GHZ)	0.094
Static Power (W)	0.074
Dynamic Power (W)	0.41
Total Power (W)	0.484

Lab7.2	
LUT	2275
DSP	3
Memory (BRAM)	34
Flip-Flop	1762
Latches	0
Frequency (GHZ)	0.105
Static Power (W)	0.074
Dynamic Power (W)	0.372
Total Power (W)	0.447

Fig.17 Statistic for Lab7.1 and 7.2

The design in Lab 7.2 is way more efficient than 7.1. In side Lab 7.1, we are using incredible amount of registers, which cause a high usage for flip-flops (for storage) and LUT (for selecting Mux). In lab 7.2 however, we replace the register memory with on-chip memory, which is a separate part from LUT and flip-flops and does not consume other resources on chip. In 7.1, even though we did not use on-chip memory, it still exits on the FPGA, we are essentially wasting that resources. We could verify this by looking at the higher frequency of Lab 7.2.

The first tradeoff in for Lab 7.2 is the complexity of read&write&handshaking process compare to straightforward Lab 7.1. Second tradeoff is we have to consider the latency of read&write for on-chip memory, while registers does not have latency.

Conclusion

The functionality described in the lab manual are all successfully implemented in this lab. The lab 7 provides many useful techniques for the final lab. For example, now we have a method to assign different colors using a palette for different characters. And we learned how to use block memory for a system on chip design.

The lab manual for 7.1 is clear, however the part for 7.2 is not. We spend a lot of time figuring out what we need to do within that AXI module. And I think it would be better to give more illustration on this part.