Lab Report #7

Aryan Shah (aryans5), Dev Patel (devdp2)

University of Illinois at Urbana Champaign

ECE 385 - Digital Systems Laboratory

Professor Zuofu Cheng, T.A. Gene Lee

November 7, 2023

Introduction

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

The HDMI interface allows for the transmission of high-quality video and audio signals across various devices, typically an input device and a display device such as a monitor or TV. The way HDMI works is through cables that carry both audio and video data (using various communication protocols and encoding) in a variety of formats. This allows for simplification, ability to output high-quality audio and video, as well as being compatible with most modern-day devices. The purpose of our design was to create a simplified text mode graphics controller. This controller is connected to the AXI4 memory-mapped bus, a communication protocol in which different hardware components communicate with each other by reading from and writing to memory-mapped addresses within a single bus' address space. The design also allows for the support of 80 column text mode via output to an HDMI. Next, we extended our original monochrome text display with the addition of video memory and color support for the purposes of displaying color text. Our overall design involved the use of creating sprites that were pulled from ROM data. The way we integrated this was accessing the address of our sprite, the location of the sprite (on our display), and its corresponding color scheme. All this data was stored in a control register, and the HDMI interface allowed us to display this in high-quality color to an external monitor.

Address how the design you created builds on top of the basic one provided for Lab 6.2

Lab 6.2 was somewhat similar in the sense that we used a VGA-HDMI IP to display a collection of backgrounds involving text and sprite components of various colors. However, its main difference was that the information stored was not within registers (Lab 6.2

implementation), rather on-chip memory using ROM (read-only memory). Using the soft-core microprocessor MicroBlaze, Lab 6.2 involved the display of a ball on a monitor which can be controlled via keypresses on a keyboard. This MicroBlaze was used to facilitate read/write communication between registers that stored our input information (transmitted through SPI protocol). Similarly, Lab 7 also involved the use of our HDMI interface via the same VGA-HDMI IP, but we had to implement text on a screen from specified colors of our on-chip memory, which therefore required the use of a memory-mapped AXI4 interface.

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

The primary method for hardware to software communication utilized in Lab 6.2 were keycode case statements that we hardcoded into our ball.sv file. In Lab 7, however, we created dedicated IP cores (such as hdmi_text_controller) as our communication method. The disadvantages of our new approach come from some of our requirements in Lab 6.2. Since the design for our Lab 6.2 was simpler (only 4 keycodes needed to be hardcoded to move up, down, left, right) the keycodes provided a less complex design. It was easier to change and test (modifying ball.sv code and re-running implementation as opposed to editing a packaged IP and upgrading it in our block design). The advantages of our AXI4 interface include the ability to pull from on-chip memory. Registers are typically smaller (which was fine for our Lab 6.2 limited data), but larger-scale projects such as lab 7 could benefit from the increase in size capability and ability to store more complex data structures (images, video, audio) in on-chip memory. Since on-chip memory is read-only, it is also a non-volatile method of memory (information stored even when the system is shut off), allowing for safer retention of data.

Overall, for larger and more complex applications, the access of data from on-chip memory has more benefits as opposed to storing in registers.

Written Description of Lab 7 System

Week 1 (Monochrome Text Display)

Written Description of the entire Lab 7 system

Beginning with lab 7.1, our preliminary design included the creation of a monochrome (black/white) graphics controller. From the given code, we had to increase the amount of registers utilized to 601, 600 of which contained our glyphs, composed of VRAM data to display symbols (sprites = text glyphs), with the last being used as our control register. This control register determined what color would be outputted onto our HDMI display. Our color algorithm was a fixed function that was later modified in lab 7.2 to pull from on-chip memory. The given specifications for our screen were confined to a 80x30 grid (2400 total characters). Each component in our grid is a 8x16 pixel space, each containing a glyph. When running, the pixel data would be pulled from our 600 registers (each composed of 4 characters that need to be drawn, 600*4 = 2400 total characters as specified; an invert bit determines if the color should be inverted between black/white) and our 32-bit control register contains the necessary background colors for each pixel space. Lab 7.2 (discussed later) involved the extension of the lab to support multi-color output, allowing us to pull color data from on-chip memory using ROM as opposed to registers.

Describe at a high level your HDMI Text Mode controller IP

The HDMI Text controller we designed facilitated communication between our AXI4 and the HDMI display. The IP has features such as reading and writing of our 601 registers (glyph

and background color information stored within them) as well as our inverse bit to differentiate between black/white characters. Within this IP, we also define several SystemVerilog modules. These include the given font_rom and vga_controller files, that were used to instantiate the AXI4 memory-mapped interface, font_rom containing font data for our glyph display, and our vga_controller allowing for output onto a VGA display (which we further convert to HDMI).

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

The HDMI AXI registers were read from and wrote into utilizing our hdmi_text_controller and hdmi_text_controller_AXI modules. We extended the variable 'local_reg', a 32-bit array, to handle the case of 601 registers as needed. Inside an always_ff block in the code, we make use of conditional statements. If our Urbana board's reset button is pressed, all 601 of our registers are cleared. To clear our registers, we set the data from the VRAM to a value of 0. After checking for the reset case, we then set up a conditional to see if our CS (chip select) is '1'. Reading and writing operations are permitted if our chip select is high. If a CS is asserted for either a read or write operation, its corresponding 4-bit BE (byte enable) signal is then checked. Byte enable values are associated with different actions as specified in our experiment manual. If our read bit is higher than our write bit, it indicates that we must read a value from the register rather than write to it. Therefore we hold a data read variable that is set to the value of our local_reg at the necessary address.

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 vga_controller SV file contains the two variables, DrawX and DrawY that aid us in drawing the text characters onto our display. These variables are initially set to the upper-left

corner of the glyph that is being drawn. We first divide the value of our DrawX by 8 and DrawY by 16 (each glyph is a 8x16 pixel chunk) to get the number of pixels that the glyph is drawn within. Our next step is to calculate the byte address using the equation: rows * cols + curr_col. Plugging in our actual variables, the line of code resembles (80 * DrawY/16 + DrawX/8), noticing that we can only access data one row at a time and our data is stored in row-major order. Our last step to get the correct index is to multiply the byte address by 4, as each register contains 4 characters.

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

The inverse color bit allowed us to achieve a monochrome (black and white) display. It indicates which bits need to be reversed to white, allowing them to be readable on an all-black background. The way it works is that the bit can store a value of either 1 or 0. If its value is 1, the pixel is inverted from its original value specified by the font_rom data. This value then goes to our control register to determine its final color. The control register's purpose is to determine the final color of background and other object colors. Using the given VRAM bit encoding values, we translate pixel values (from our above calculations) into R,G,B signals that then output monochrome characters onto our display as specified.

Week 2 (Color Text Display)

Describe the hardware changes you had to make to support the use of multi-color text. At the minimum, you must describe:

As will be mentioned, the week 2 portion of the lab challenged us to support multi-color text. With this change comes the necessity of on-chip memory as opposed to register-based memory. The following responses will be related to what was done to accomplish this.

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

During our week 2 portion, we pulled stored information from on-chip memory due to its various benefits mentioned earlier. In short, due to the amount of memory space needed to convert monochrome text into multicolor text, it required the use of on-chip memory-based VRAM as opposed to register-based VRAM. Our first change to support multicolor was that each character had to be extended to 2 bytes. The 2nd byte added contains two 4-bit color codes that index to the necessary color from the color codes in our color palette file. We therefore had to instantiate a BRAM to our IP hdmi_text_controller module. The data for the colors could now be pulled from the random access memory in this BRAM rather than from registers. The color palettes remain in register data as they were in week 1.

Corresponding modifications to the IP Editor

The modifications needed to the IP block included the width of the address extended from 11 bits to 12 bits to accommodate for our VRAM. The rest of the logic remained the same.

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

Due to the fact that our bytes hold 2 characters instead of 4, we multiply our byte address by 2 instead of 4 accordingly. We first utilized the DrawX and DrawY signals, which point to the upper left corner of the text, to calculate the row and column that the glyph is drawn at. Since we have a 8x16 space within the 80x30 grid for each glyph, we divided DrawX by 8 and DrawY by 16 which resulted in the correct pixels that each glyph draws within its space. Using our row-major order equation: 80 * DrawY/16 + DrawX/8, we can find our byte address value. The word address in the VRAM (to index) is this byte address value multiplied by 2, as specified.

Any additional modifications which were necessary to support multicolored text

Each register we created had to support 2 colors, rather than 4, as mentioned.

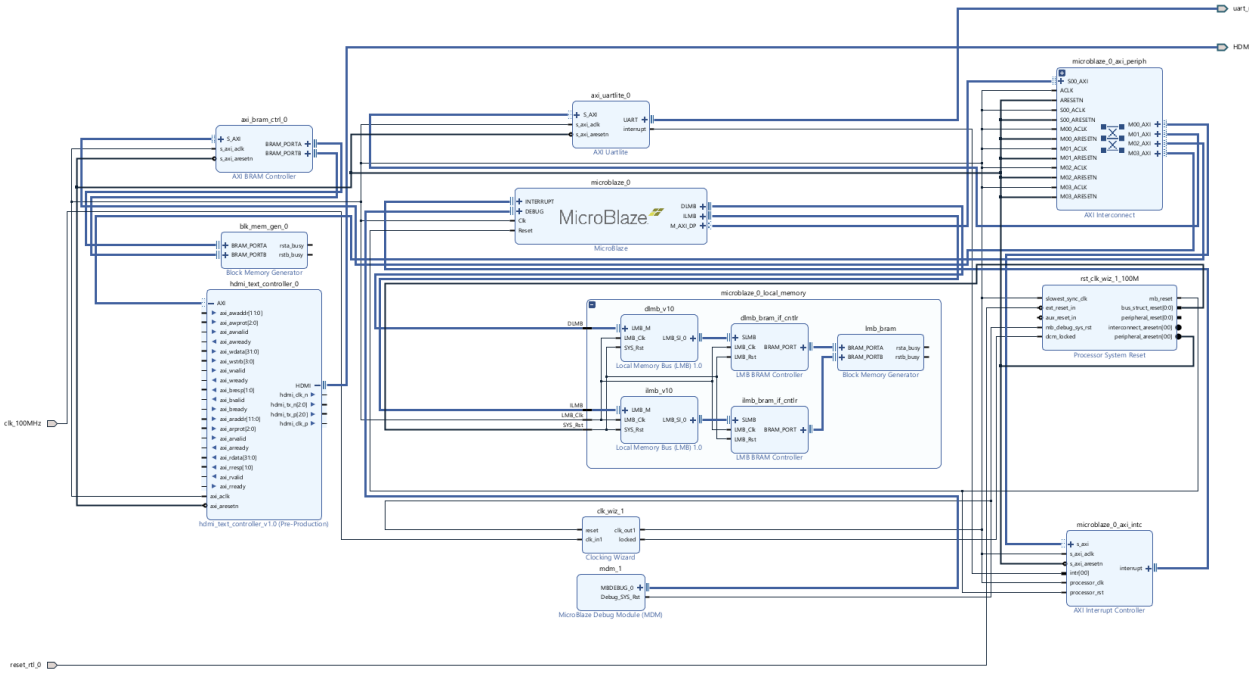
Additional hardware/code to draw palette colors

We created color palette registers to hold 16 various colors that can be outputted. We reused the same logic from lab 7.1 for our color palette, with the exception of adding a check to see which color palette is read/write from (to account for multiple colors).

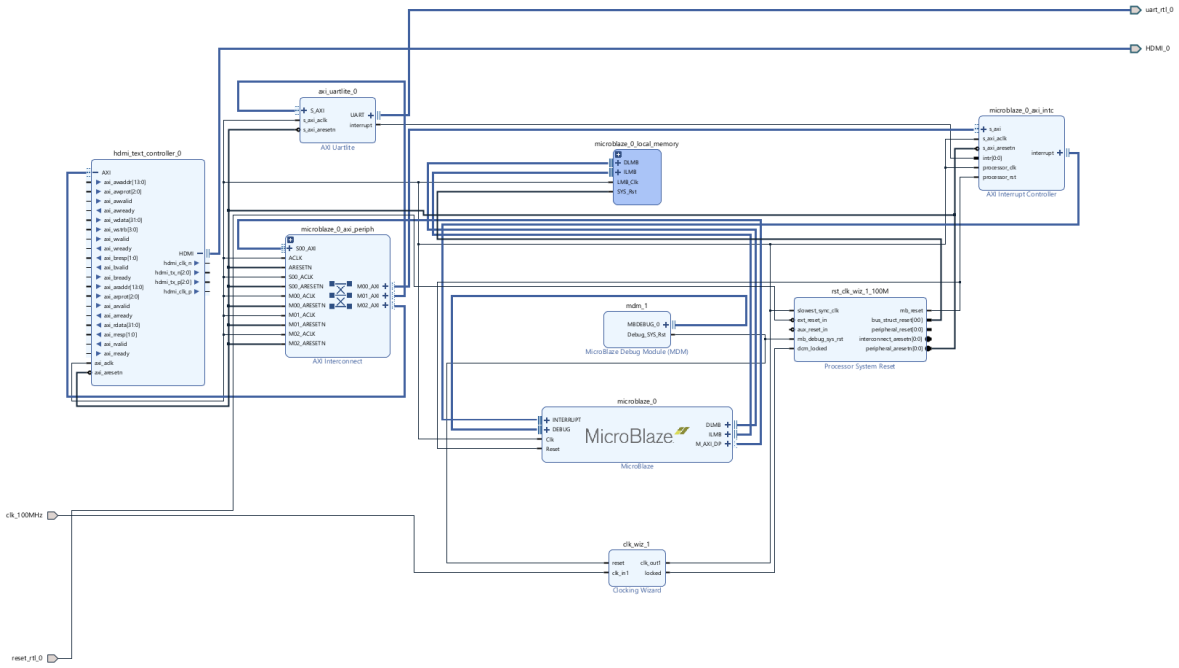
Block Diagram

Block diagram of the IP you designed for each week

Week 1



Week 2



Module Descriptions

Week 1

Week 2

Design Resources and Statistics

Week 1

LUT	14338
DSP	3
Memory (BRAM)	32
Flip-Flop	21129
Frequency	137.816979 MHz
Static Power	0.074 W

Dynamic Power	0.408 W
Total Power	0.482 W

Week 2

LUT	1978
DSP	3
Memory (BRAM)	34
Flip-Flop	1720
Frequency	113.752702 MHz
Static Power	0.381 W
Dynamic Power	0.408 W
Total Power	0.445 W

Briefly discuss the difference between using on-chip memory for VRAM and registers.

Registers are high-speed memory locations located within a computer's CPU. When information is pulled, the CPU must actively process the data and all its related information. They are typically very fast and used for simpler designs where the necessity for large data storing is not needed. On-chip memory, on the other hand, is memory that is stored within a dedicated GPU system. It is typically used to store render-based display data, something that continuously needs to be displayed. Therefore, it is important that this memory must be stored in a place separate from the CPU, such that it can be accessed at all times. When the main computer is shut down, the CPU cannot be accessed whereas on-chip memory can. Furthermore, on-chip memory is more efficient for larger-scale applications with more amounts of data that need to be stored, such as support for multicolor text as opposed to two (black and white) for monochrome.

Which design is more efficient, what are the tradeoffs?

The design utilizing on-chip memory is more efficient. Tradeoffs include the necessity for speed over capacity or vice versa. Registers offer much faster speed, but limited storage which was more important to support multicolor text in week 2. Slower accessing is therefore a tradeoff of on-chip memory.

Conclusion

Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.

We had no design-related errors that could be attributed to improper understanding of the material. It was mostly the time-consuming portion of this lab that took a while. Before our demo, we had some unfinished logic in our color palette and C files that were attributed to our lost points. These were quickly resolved according to the lab specifications afterwards.

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

This design, along with the incorporation of concepts from our other labs could be very useful for our final project. Our proposal involved the design of a PacMan game. Being able to use on-chip memory-based VRAM to support multicolor text would be essential for certain features we would like to add such as multicolor background and objects as well as the HDMI interface to display it. Extensions of this design could also be similar to our final project idea, including many popular arcade games such as Snake, Pong, Tetris, etc. Leaning heavily on concepts from Lab 6 and Lab 7, we hope to make a fully-functional PacMan game displayed on an HDMI monitor with features that challenge us to learn new concepts, such as multiplayer, sound components, AI, etc.

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.

N/A, the lab materials were sufficient to carry out this experiment. It is important to note that we were unable to finish our design in time (even with a 1 day extension) for the demo, therefore having to finish it onwards for the purposes of this report and final project. This could be attributed to the fact that both of us had multiple midterms that week. However, the consensus on CampusWire suggests that we were not the only ones. For future semesters, it may be beneficial to extend the period of time students are able to work in this lab.