ECE 385

Fall 2023

Experiment 7

HDMI Text Mode Controller with AXI4 Interface

Cher Rui Tan, Adithya Balaji
Section: Friday 11.30am
TA: Gene Lee

Introduction

For this lab we built a HDMI text controller that is able to output text with color to act as a text to graphics controller. In lab 7.1, this entailed using 601 registers for the VRAM to be able to output glyphs on the screen, and in lab 7.2, this was expanded to accommodate 2^16 colors using block memory instead of registers. The output text on the screen will contain the information on the color of the text and the background of the text, and the colors would follow this.

In lab 6.2, a basic design was incorporated using a GPIO to connect the MAX3241E to the microblaze, to output a controllable ball to the screen. On the other hand, for lab 7, we did not use any GPIOs, rather we had a handshaking procedure in a separate AXI module to control reading and writing into registers that contain the information pertaining to the glyphs and colors. Additionally, we used a set color palette function to set the colors to be used in the drawing logic.

For the design of Lab 7.2, we output text of multiple colors to the screen, and we do this through the HDMI Text controller IP module that we create that handles reading and writing of data from/to block memory, to determine the character and the color of the character and its background to the screen using a VGA to HDMI.

The advantages of the Lab 7 approach is that the IP is transferable between platforms compared to lab 6.2 where we instantiate modules within the code and use GPIO. The GPIO is platform-specific and may not exist outside Vivado. Furthermore, the IP approach allows us to encapsulate our logic within a reusable IP module, that can be reinstated for other projects.

2. Written Description of Lab 7 System

a. Week 1 (Monochrome Text Display)

i. Written Description of the entire Lab 7 system

At a high level, the graphics controller created in week one supports a 30x80 rows and columns structure for a total of 2400 characters, which are any one of the 128 glyphs from the IBM code page 437. This entails having an 8 bit string to specify a character, with 7 bits to represent the 128 glyphs and a control bit to specify whether to draw the glyph to have inverted colors.

The provided bitmap specifies that each character consists of 8 x 16 pixels, and taking into account the 30 rows and 80 columns, the output display will have the same 640 x 480 pixels as lab 6.2.

The VRAM is created using 601 registers, with 600 registers to store memory addresses corresponding to 4 characters each (to make up 2400), and a control register for determining foreground/background colors as well as inversion, using the logic presented in the figure below.

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 VRAM is memory mapped to the AXI bus, which allows the MicroBlaze to access and modify the contents of the VRAM such that the software portion handles the text and color being displayed.

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

The HDMI text controller takes in the data from the AXI Bus and stores it in the VRAM for usage. This data is accessed based on the x and y coordinates of where the cursors are drawing at the moment. This allows for the glyphs to be drawn and with what color pixel. This color determination is done in the color mapper where the color of the foreground and background are pulled from the 601 register.

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

The read and write for the registers was pretty straightforward. After instantiating the registers as an unpacked array, we used each register location to store 4 characters. To write, we simply omitted the last 2 bits of the address to look for the right register, then using the last 2 bits of the address as a secondary address, we wrote into the corresponding right 8 bits of address. To read from the address, we used the DrawX and DrawY coordinates to determine which register, then which word we were addressing. This allows us to access the right bits for reading.

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

The algorithm was as follows: address = DrawX/16 + DrawY/32*80, this gives us the register which we need to access and the specific word in that register. The last 2 bits of the address would be the position of the word in the address and the bits [11:2] are the register address. Grabbing the correct 8 bits with bit slicing, we assign the most significant bit as the invert bit and the next seven bits as the address of the character glyph.

With this address, we index in the fontRom to the start of the character glyph by taking character address*16 + DrawY%16. This allows us the ability to index through the frontRom character we have chosen based on where the DrawY is on the screen. We read from the frontRom and the sprite_data is reversed with [7-DrawX[2:0]] to ensure little endian ordering. We also directly access register 601, the control register to get the foreground and background color.

With the colors, the data from the fontRom and the inverse bit we make case statements which draw the entire screen.

b. Week 2 (Color Text Display)

To enable the multi color text, we have the addition of a color palette made of 16 registers. Furthermore, we changed the VRAM to our on chip BRAM with 1200 memory locations, this taking up much less space on the board as compared to the registers. Furthermore, we modified the BRAM to enable dual channel read and write. This allowed us to use one memory port purely for communication with the AXI while the other memory port was used to interact with the color mapper and fontrom.

Corresponding modifications to the IP Editor

The key differences between week 1 and week 2 in the IP editor was changing the number of registers from 601 to 16 to support color indexing, as well as the instantiation of a 1200 deep 32 bit wide Block memory RAM to store and access the information instead of using a register. This resulted in the address width changing from 12 to 16 allowing us to address up to x201F.

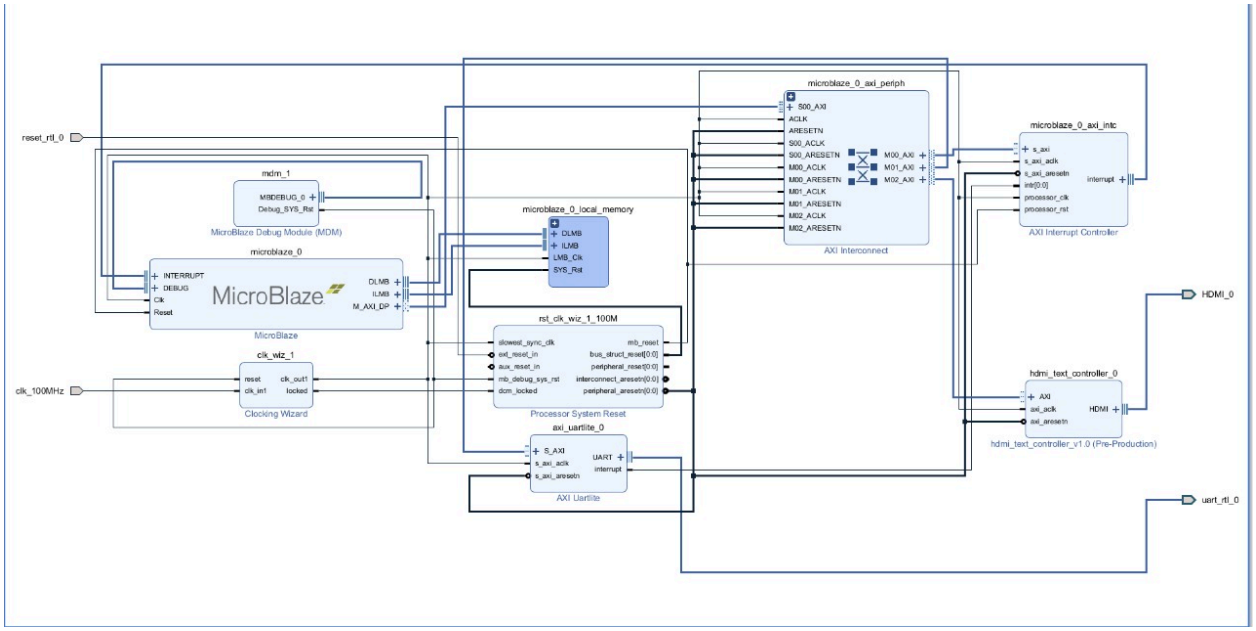
Modified sprite drawing algorithm with the updated indexing

In week 2, 2 characters are stored in a register unlike the 4 characters per register in week 1, so we changed the indexing portion of the algorithm to determine what to access, by dividing by 2 instead of 4. Also, only one bit was needed in week 2 to decide which character is being accessed.

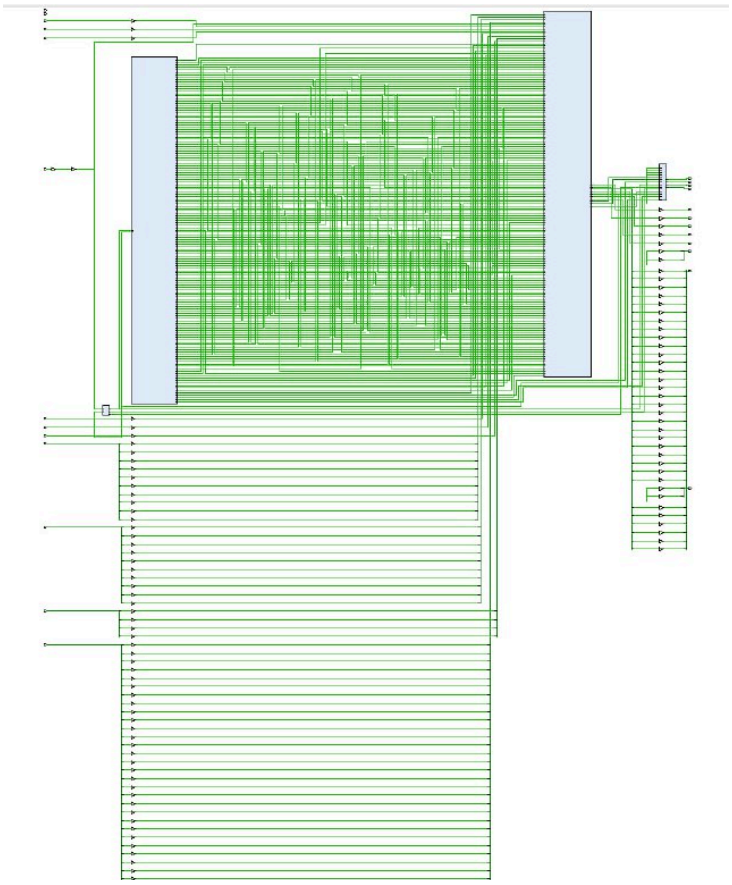
Any additional modifications which were necessary to support multicolored text and Additional hardware/code to draw paletted colors.

We had additional logic to set the color palette in the C code, to be used in the hardware implementation. The BRAM would contain 32 bit addressees which not only contains the character information but also the color index, including a foreground and background index for each of the 3 RGB colors as well as inversion bit. Additionally, we needed to retrieve the relevant 12 bits for each character and send it to the color mapper where the logic to set the RGB values were done to output the text with correct color to the screen. Furthermore, we have logic to distinguish if we are writing and reading into the palette region or the BRAM region.

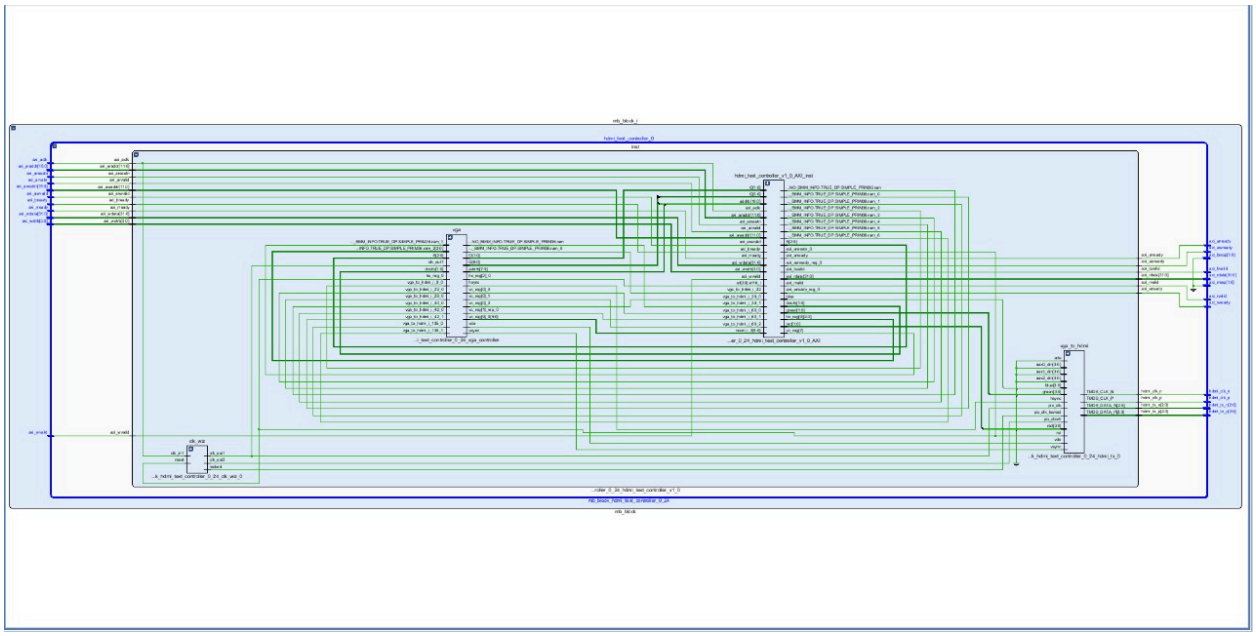
Block Diagram



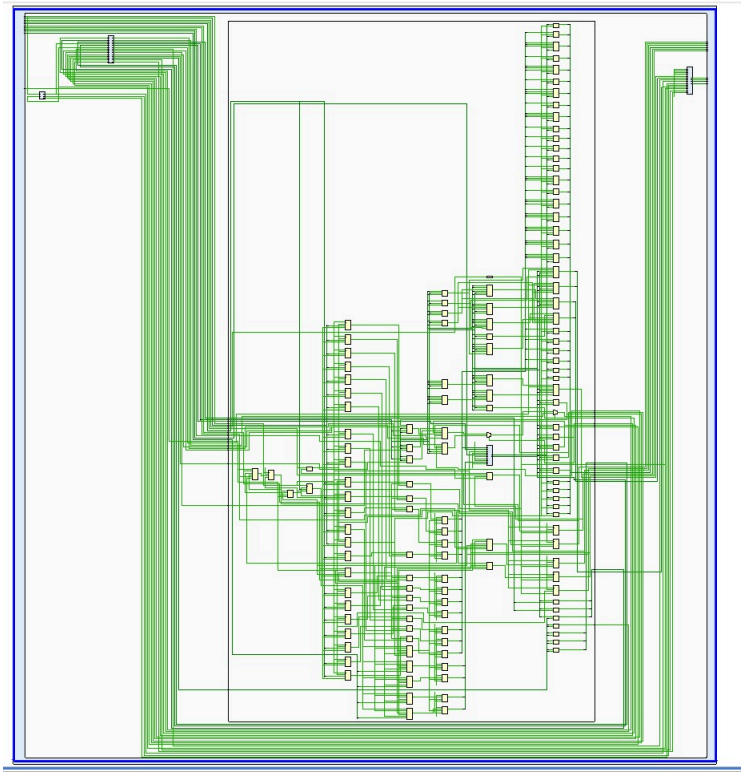
Top level diagram



Week 1 HDMI text controller



Week 2 HDMI text controller



Week 2 HDMI Text Controller AXI

Module Descriptions

Microblaze
Creates a basic microblaze block diagram with its associated peripherals, with a microcontroller preset which is relevant for this low-level program.

AXI GPIO
Serves as a link between the AXI bus as well as input/output pins on the FPGA, used to read and write data from/to the FPGA. Allocate and maps memory in the Microblaze for different peripherals

AXI Uartlite
Universal asynchronous receiver transmitter, allows for asynchronous UART communication via the AXI bus, handling data transmission and reception. Used for communication with microcontrollers.
AXI Interconnect
Allows for connection and transmitting data between multiple AXI modules, by managing data traffic by handling routing and latency control. Serves to make the system more efficient and flexible.

Clocking wizard

Takes a 100 Mhz Clk input and outputs the share clock to the system

Processor system reset
Provides customized resets for the entire processor system and processor, as well the interconnect and the peripherals.

Microblaze debug module
Enables JTAG-based debugging for Microblaze processors.

AXI Interrupt controller
Concentrates multiple interrupt inputs from peripheral devices to a single interrupt output to the system processor.

Hdmi text controller
This is the main IP block we modify in the lab, which contains the BRAM and the handshaking process AXI module, as well as the other modules to ensure the text converter is functional

Internals of HDMI Text controller AXI (Week 1)
In the first week, the AXI comprised 601 registers of 32 bit length. This was instantiated with an unpacked array.

Internals of HDMI Text controller AXI (Week 2)
The 601 registers were replaced with 1200 deep on chip BRAM. It also contains 16 registers which comprises the color palette. A mux is instantiated to decide between the two when both reading and writing.

Design Resources and statistics

The design resources and statistics for each week are shown below:

Lab 7.1
The frequency is calculated by doing 1/(10ns-wns);

LUT	15235
DSP	3
Memory (BRAM)	32
Flip-Flop	21228
Latches*	0
Frequency	0.100MHz
Static Power	0.402
Dynamic Power	0.077

Total Power	0.479

Lab 7.2

The frequency is calculated by doing 1/(10ns-wns);

LUT	2477
DSP	3
Memory (BRAM)	34
Flip-Flop	1618
Latches*	0
Frequency	0.116 MHz
Static Power	0.074W
Dynamic Power	0.379W
Total Power	0.454W

Week 1's design with registers is more efficient in terms of accessing speed, however the trade off comes in the board space which would not have been possible with 1200 registers. The time taken to place and route as well as the power consumption. Week 2's design which used the BRAM has a much faster place and route and it has a set amount of board space already used up.

This can be seen from the Week 1's design taking a great number of LUTs at 15235 as compared to week 2's 2477 and flip flops decreasing from 21228 to 1618 due to most of them being used for register instantiation. Furthermore, the larger amount of LUTs results in more power usage as seen from the greater amount of power in Week 1 at 0.479W as compared to 0.454W in week 2. The registers also result in a longer place and route with the frequency decreasing. This is the result of the renovation of the 601 sized MUX which would cause combinational logic delay.

The BRAM increased from 32 to 34 as we instantiated an on chip dual channel BRAM which was 1200 deep and 32 bits wide.

Conclusion

We faced issues with the color mapper, where we were cutting off the first 3 bits of our color code resulting in the lack of red in our screensaver as well as a little green. We hoped to correct the issue by locating the bit truncation and fixing it.

In conclusion, we learn how to use the color mapper, allowing for variable colors in our screensaver. We hope to incorporate this in our final project if we decide to do multi color diagrams or figures which move.

We think that lab 7.2 barely had any documentation and felt a little bit counterproductive since most of the time was spent figuring out how to do something instead of coming up with the logic.