ECE 385

Fall 2023

Lab 7: HDMI Text Mode Controller with AXI4 Interface

Alex Yu, Corey Yu
GL/ 13:45-14:00
Gene Lee

1. **Introduction**

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

        In this lab, we are trying to make a HDMI text controller. This text controller will read input from the AXI bus, store the data into the register, implement the draw logic, and finally, output it onto the screen. Our goal is to make an IP that is able to read from the AXI bus and draw different colors of text and background on the screen.

    2. **You should address how the design you created builds on top of the basic one provided for Lab 6.2.**

        For lab 6.2, we drew the ball onto the screen, and the ball was able to be controlled by the keyboard. In lab 7, we reuse the file color mapper, vga controller, and we modified the existing microblaze from lab 6.2. These two labs have similar logic of how they draw on the screen, which is the part of the VGA controller and the color mapper, but different ways to get the data input. For lab 6, we use the MAX3241E and the USB as the input, but in lab 7 we use the AXI bus to get the input from software that runs on the microblaze.

    3. **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.**
        6.2 utilized the MAX3421E to communicate to a USB keyboard through quad SPI. Volatile pointers that correspond to the address of the GPIO for the is simpler
        What is diff between AXI and SPI isn't axi also comm protocol: AXI interconnect is internal only, SPI is to talk to external
        In 6.2 the software is purely an intermediary, the lab is hardware based. The MAX3241E reads the keycode which is connected to the quad SPI which the AXI-lite bus interfaces with. The software defines communication of the AXI bus and SPI, and the resulting keycode data is fed to hardware which handles the rest of the drawing logic. In 7.2 the software tells the hardware what to do. The IP block serves as generalized hardware and the software program (hdmiTest) can be changed to draw whatever text is desired.
        Lab 6 is simpler as you avoid burning an entire day fixing Vivado errors when

creating a custom IP. Lab 7 is far more portable/flexible, as you can merely drop in the IP and program whatever text you want drawn in software.

2. Written Description of Lab 7 System

- **Week 1 (Monochrome Text Display)**
  i. Written Description of the entire Lab 7 system

  The lab 7 system consists of a standard microblaze based SoC with standard peripherals: clocking wizard, AXI interconnect, interrupt controller, etc. In addition, the system includes a custom HDMI text controller IP that supports character addressable text graphics (the screen is 80x30 characters)
  ii. Describe at a high level your HDMI Text Mode controller IP.
  The HDMI text controller IP wraps the entirety of the System Verilog needed to display text (e.g. the VGA and color mapper that were part of the top-level block diagram in lab 6.2). It handles communication with the AXI bus, the logic to access the correct text, defining RGB values for a given pixel and contains the VGA and VGA-HDMI modules to output the appropriate HDMI signals for a given pixel.
  iii. Describe the logic used to read and write your HDMI AXI registers.

  The logic to read the HDMI AXI register is defined in the hdmi AXI.sv. This defines the AXI-lite handshaking protocol the memory-mapped peripheral uses. The AXI4-Lite interface consists of five channels: Read Address, Read Data, Write Address, Write Data, and Write Response. All five transaction channels use the same VALID/READY handshake process to transfer address, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate at which the information moves between master and slave. The information source generates the VALID signal to indicate when the address, data or control information is available. The information destination generates the READY signal to indicate that it can accept the information. The handshake completes if both VALID and READY signals in a channel are asserted during a rising clock edge. Below is an example and description of an

AXI read.

Below, the sequence for an AXI4-Lite read is shown:

1. The Master puts an address on the Read Address channel as well as asserting ARVALID, indicating the address is valid, and RREADY, indicating the master is ready to receive data from the slave.
2. The Slave asserts ARREADY, indicating that it is ready to receive the address on the bus.
3. Since both ARVALID and ARREADY are asserted, on the next rising clock edge the handshake occurs, after this the master and slave de-assert ARVALID and the ARREADY, respectively. (At this point, the slave has received the requested address).
4. The slave puts the requested data on the Read Data channel and asserts RVALID, indicating the data in the channel is valid. The slave can also put a response on RRESP, though this does not occur here.
5. Since both RREADY and RVALID are asserted, the next rising clock edge completes the transaction. RREADY and RVALID can now be de-asserted.

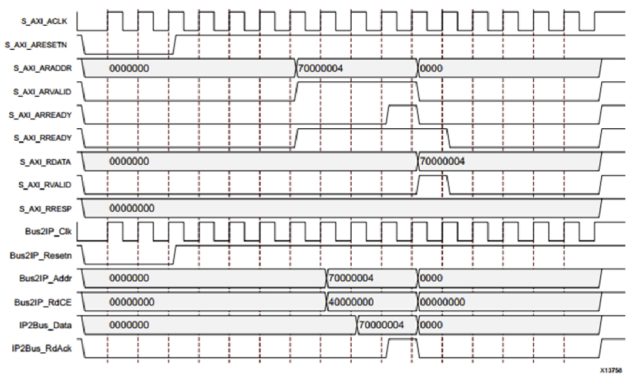


Figure 5 - AXI4-Lite Read Transaction

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

1. spongebob <= slv_regs[(DrawY/16)*20+ DrawX/32];

2. temp <= spongebob[((DrawX % 32)/8)*8 +: 8];

3. DIS_INVERT <= temp[7];

4. fontrom_food <= (temp[6:0] * 16) + (DrawY % 16);

5. ctrlregs <= slv_regs[600];

DrawX and DrawY correspond to the pixel that is currently being drawn. Each character is 8x16, so DrawX/8 and DrawY/16 convert from the pixel coordinates to character coordinates Cx, Cy. The screen is 80x30 characters so to flatten the 2D row-major character index to 1D, Cn = Cx + Cy *80. Each register holds 4 bits ; as such the least significant 2 bits correspond to which portion of the register.

Thus the desired register is Cn[11:2] or Cn/4 which is fed into slv regs. This is line 1
The character number was encoded in the register and extracted. There are 32 pixels in a register, drawX % 32 corresponds to which pixel in the register it is at, /8*8 because integer division truncates to the nearest multiple of 8. Thus line 2 slices the register to the 8 bits of the character index + invert bit.
Line 3 grabs the invert bit as it is always the MSB of the 8.
There are 16 rows per character; the character index*16 tells you what line of fontROM to start at and adding DrawY%16 as you need to go from that line to that line + 15 of fontROM.
Line 5 grabs the control register which encodes color data.

Ctrlregs, dis-invert and fontrom_food are passed into the color mapper file. Fontrom_food[7 - DrawX[2:0]] is checked and the appropriate color (foreground or background from the control register, flipping them if dis_invert = 1 [the foreground/background should be flipped then]) is assigned to R, G, B outputs. Fontrom_food[7 - DrawX[2:0]] is checked rather than Fontrom_food[DrawX[2:0]] because the data is little-endian and needs to be reversed to print correctly (otherwise the characters would be backwards).

Week 2 (Color Text Display)

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

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

  Compared to week 1, for week 2, we use the BRAM and the register together instead of just the registers. In 7.2, we use BRAM to record what we are going to print on the screen, and the 16 extra registers as the color palette to track the 16 different colors that can be on the screen at the same time.
  You only need to read/write to one register at a time so the limited ports were not an issue so long as you have two ports available (which was why a dual port was used).

- **Corresponding modifications to the IP Editor.**

  In order to modify it to the BRAM implementation, we have to change the existing 601 registers to 16, since we only need 16 now for the color palette. We change the address width from 12 to 16. The reason why it

was 12 for the first week is we want to access all 2400 addresses on the screen, and for the second week we need to access the register which is at x201F, to check if we are accessing BRAM or the palette, so we change it to 16. Basically, we had to add in the logic that distinguish between the palette and BRAM for both read and write now.

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

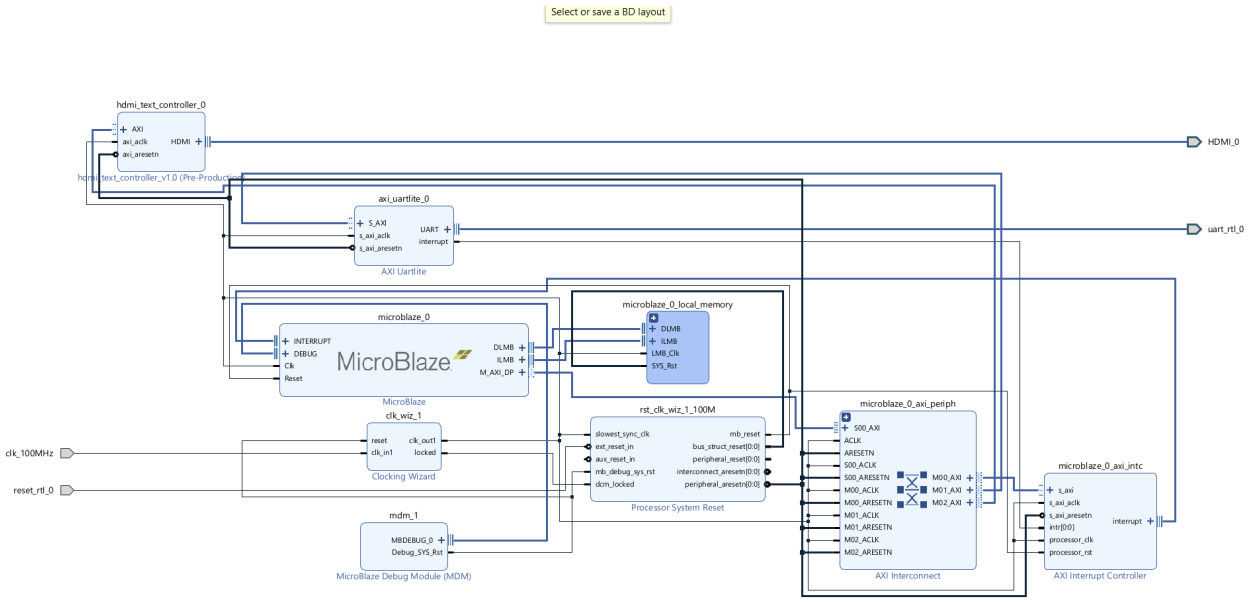
  In week 1, we store 4 characters per register, but in week 2, we only store two characters per memory location. This means we have to modify our algorithm from dividing all the characters by 4 to dividing them by 2. Also, now we only need 1 bit instead of 2 bits to decide if it is the first character or the second character in one memory location.

- **Any additional modifications which were necessary to support multicolored text. Additional hardware/code to draw paletted colors.** (We combined the answer for these two.)

  Now every character can be drawn in its own background and foreground color. The 32 bits in the BRAM now also contain the color index for the specific character. The index is for the color palette, this way we can use the 4 bits to access the color palette to get the foreground color and the background color for the character. To implement this design, we also had to change what we call the color mapper, sending in a 12 bits value to the color mapper, and the R, G, B will access the 3 groups of 4 bits in order.

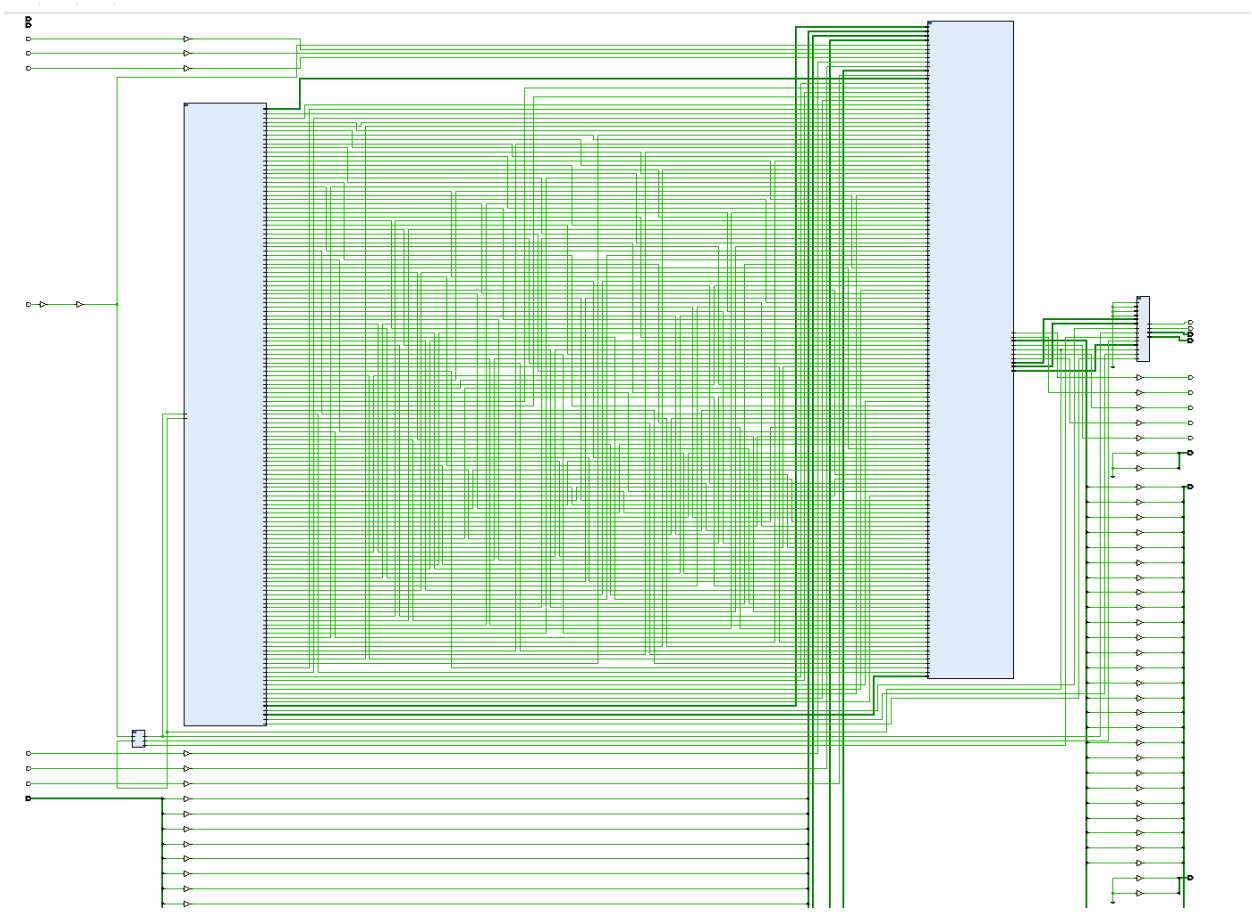
3. Block Diagram

a. This diagram should represent the placement of all your modules in the top level.

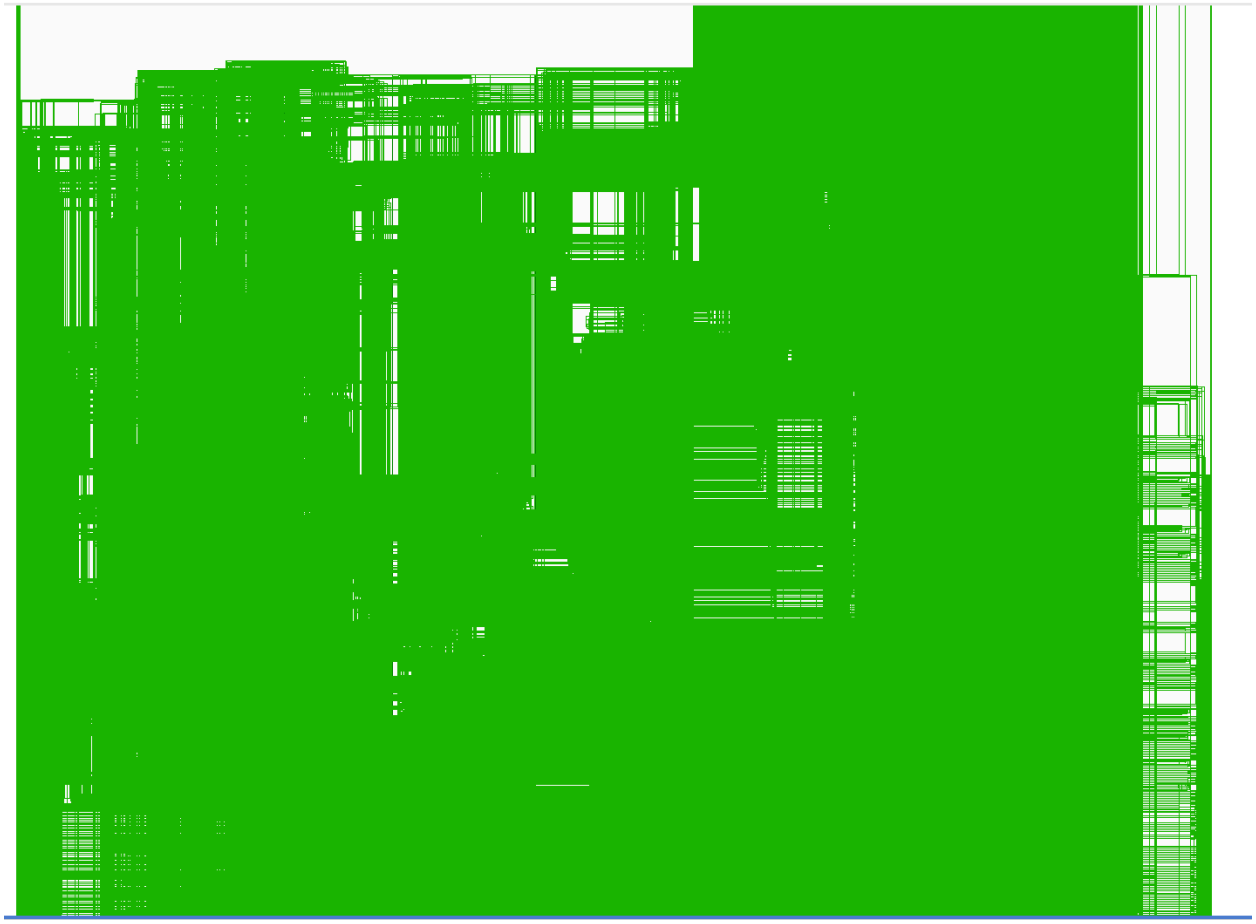


b. Internal block diagram of the IP you designed for each week.

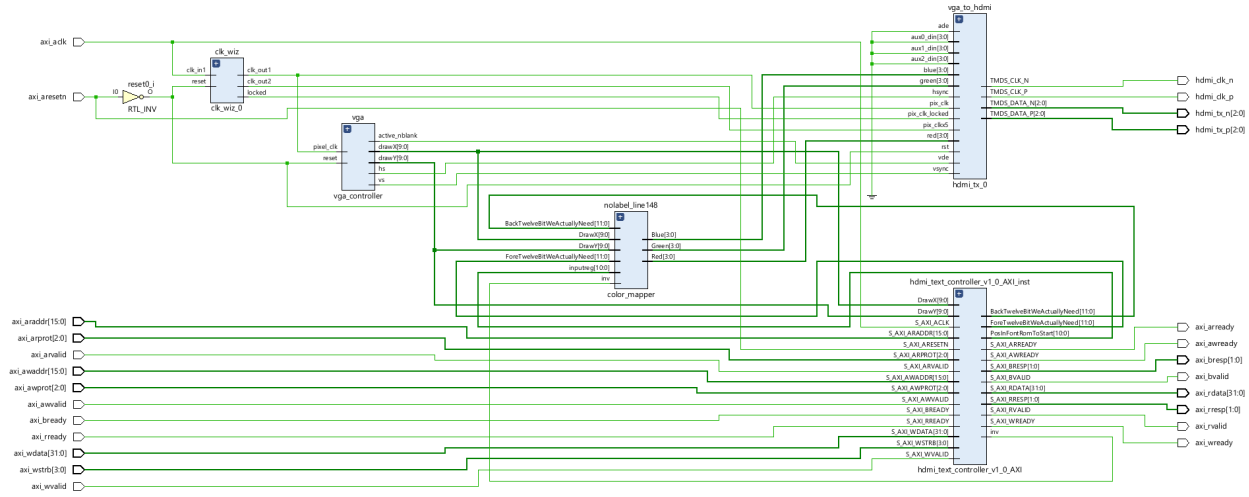
Lab 7.1 hdmi_text_controller IP block diagram



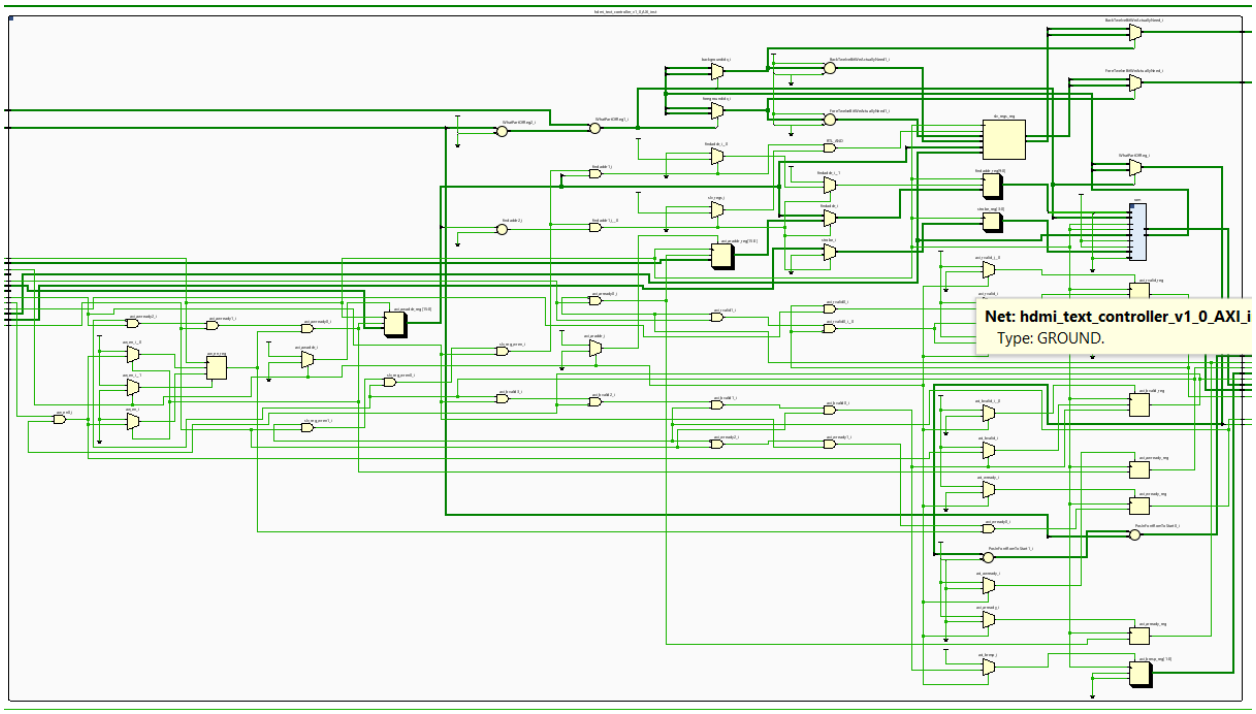
Lab 7.1 hdmi_text_controller_v1_0_AXI diagram(can only see wires as there are 600+ lines between different components):



Lab 7.2 hdmi_text_controller IP block diagram



Lab 7.2 hdmi_text_controller_v1_0_AXI diagram:



i. **Note that depending on your layout of the registers inside your main module, the Vivado schematic view may be illegible, in which case you should draw a block diagram using software. You may start with the provided materials (e.g., in IAXI), but you should fill in the specific signals between the modules and the inside subcomponents within each module.**

   We are able to generate the IP block diagram for week 1. The one that has 600+ wires connecting between the AXI and the vga controller. This is because we had to pass in the drawX and drawY to determine which part of the register we are grabbing.

ii. **You should have block diagrams for both the Week 1 and Week 2 portions, a good setup is to show the common components (e.g., the SoC setup) first and then show diagrams for both the Week 1 and Week 2 HDMI controller components.**

   Covered in the diagram above as we showed both IP block diagrams for both week 1 and week 2 IP.

iii. **If your design has a state machine, you should include a State Diagram as well.**

   We don’t have a state diagram.

4. Module Descriptions

A guide on how to do this was shown in the Lab 6 report outline. Do not forget to describe the Block Design file for your MicroBlaze system.

Describe the internals for each week's version of the HDMI controller IP. Note that shared components may have a single module description.

For 7.1 and 7.2, if we don't open the IP design, the diagrams look the same, with these modules:

a. **MicroBlaze:** This is the soft IP block that provides the functionality of a RISC harvard architecture processor. As described in the introduction, this is the processor that responds and processes all commands.
b. **Clocking Wizard:** This is the block that takes in the clock signal input and input the signal to all the components connected, making sure that they are synchronized. It also takes in the 100MHz and outputs 2 clock signals 25MHz and 125MHz for the HDMI.
c. **AXI Interconnect:** The AXI interconnect module can connect one or more AXI masters to one or more AXI slaves. In this case, we have multiple slaves to connect to, so we need this module.
d. **AXI Uartlite:** This module connects the processor and provides it with asynchronous serial data transfer.
e. **AXI Interrupt Controller:** This module concentrates multiple interrupt signals into one and sends it in the processor.
f. **Processor System Reset:** This module allows the user to set the parameter to enable/disable the feature. Like set it as active high reset or active low.
g. **MicroBlaze Debug Module:** It is a core that enables JTAG-based debugging of the microblaze.
h. **MicroBlaze Local Memory:** This is the memory block of the MicroBlaze.
i. **Hdmi_text_controller:** This is the custom IP we created that wraps the entirety of the System Verilog needed to display text (e.g. the VGA and color mapper that were part of the top-level block diagram in lab 6.2). It handles communication with the AXI bus, the logic to access the correct text, defining RGB values for a given pixel and contains the VGA and VGA-HDMI modules to output the appropriate HDMI signals for a given pixel.

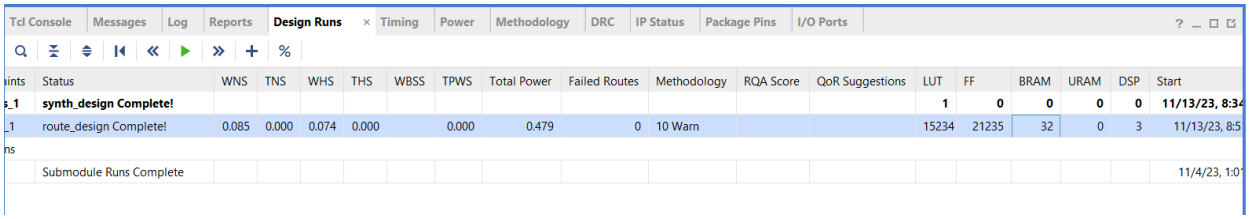
j. **AXI Timer:** This is a timer/counter interface to the AXI-4 Lite that provides the signal of AXI_reset and AXI_clk.

k. **Describe the internals for each week's version of the HDMI controller IP. Note that shared components may have a single module description.**

l. **If any modules needed to be changed between Week 1 and Week 2, you will need to describe changes here as well.**
The AXI file was changed between week 1 and week 2. Week 1 utilized the provided AXI sv file that outlined the handshaking process. In week 2 this was modified to differentiate between writing to BRAM or registers and adding a second read-only port.

5. **Document the Design Resources and Statistics from the lab manual. Each week's design should have different design statistics, and you should briefly discuss the difference between using on-chip memory for VRAM and registers. You should include both sets of design statistics. Which design is more efficient, what are the tradeoffs?**
Utilizing registers provides zero latency reads/writes, however each register has wires associated to it so place and route takes a very long time to optimize correctly. Additionally reading from "register memory" in always comb generates a very large (601 to 1) MUX which decreases the worst negative slack / increases combinational logic path. BRAM reads/writes have latency however it only has two ports (in a dual port configuration) so place and route is much faster.

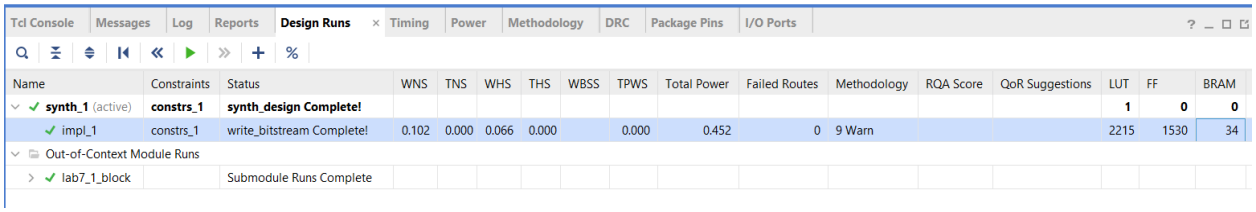
For Lab7 Week 1



	MicroBlaze
LUT	**15234**
DSP	**3**
Memory (BRAM)	**32**
Flip Flop	**21235**

Frequency	**0.10085MHz**
Static Power	**0.40236W**
Dynamic Power	**0.07664W**
Total Power	**0.479W**

For Lab7 Week 2



	MicroBlaze
LUT	**2215**
DSP	**3**
Memory (BRAM)	**34**
Flip Flop	**1530**
Frequency	**0.10103MHz**
Static Power	**0.074W**
Dynamic Power	**0.377W**
Total Power	**0.452W**

6. **Conclusion**
   a. **Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.**
   The design implements a custom IP that generates HDMI signals for character addressable text graphics. 7.1 utilized 600 registers to store character indexes and one register to encode foreground/background color. 7.2 ported character data to a 32x1200 BRAM and 16 registers for programmable color. Our 7.2 did not have full color functionality. This was caused by not having a MUX for the port a read side (the side that

connects to AXI). While port a does not explicitly need read functionality, the set_palette function implicitly performs a read when writing into the registers. Thus adding a MUX for reads the same way we had a MUX for writes would fix this.

- **What are some potential extensions of this design, what did you learn in this lab that might be useful for your Final Project?**
  The font ROM can be extended to support characters other than text. This can be used to draw simple characters for games e.g. pac man.
- **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.**
  Compile a list of common Vivado errors as we spent an entire day fighting with Vivado to build the IP in 7.1. And in 7.2 the BRAM kept getting optimized out, we restarted from 7.1 following the exact same procedure and it magically appeared.