

# ECE 385 Lab Report 7

Experiment #7:

VGA Text Mode Controller with Avalon-MM Interface

Spring 2022

Trusha Vernekar & Soham Manjrekar

Dohun Jeong (DJ)

tnv2 & sohammm2

## **Introduction**

For Lab 7.1 and 7.2, the main goal was to create a text-to-graphics controller. In the first week, we created a simple text-to-graphics controller which would output a series of texts with either green, red or blue colors. These texts would only support monochromatic colors. As for the second week, we had to create a palette that would support 16 colors instead of the 3 colors from week 1. Along with this, instead of using FPGA registers to store all the information required, we had to implement a way to store the information on an On-Chip memory, such as RAM instead. The texts on the screen would contain the name of the color for the text as well as the color for its background.

In Lab 6.2, we were given C code that communicated the keyboard inputs through USB to our FPGA to make the inputs usable by ball.sv which defined the ball's movement and position. After the position was determined, we used the Color Mapper to check if the drawn pixel will be a ball or the background. This implementation used hardware to assign and map colors to the display. For lab 7, we built on this idea by letting software handle the color assignment and mapping. The C code writes data to our VRAM, and from there the display will output a color according to the mapped color in VRAM.

## **Written Description of Lab 7 System**

### Week 1 (Monochrome Text Display)

#### *Written Description of the entire Lab 7 system*

For the entire lab, there was a lot of math involved, mainly with DrawY and DrawX values, to figure out which character needs to be printed in a specific location on the screen. Our system could support a total of 2400 characters with 80 columns and 30 rows. We had a total of 600 registers. Each register was 32 bits, storing 8 bits of information about each character. 7 bits of the information, which was a Glyph code from IBM Codepage 437. The 8th bit was the inverse bit, which would draw inverted colors for foreground or background. Since we had 2400 characters and 8 bits per character, we needed 2400 bytes of VRAM in order to save information regarding every character that needs to be printed.

Overall, for this lab, we had to read data stored in the 600 FPGA registers, find out which character needs to be printed at the specific X and Y coordinate, find the corresponding sprite character from Font Rom, and print the character to the screen by assigning Red Green and Blue

values correctly. In order to read from the FPGA registers, we needed to use the AVL bus. First, we check if AVL\_CS, which was the chip select was enabled. If it was, to choose between reading the data or writing the data, we had to check AVL\_READ and AVL\_WRITE signals. If the read was enabled, the value at LOCAL\_REG, specifically at the AVL\_ADDR location is accessed and AVL\_READDATA is set to the value from this register. As for the write signal, if this was enabled, the data stored in AVL\_WRITEDATA is placed into the LOCAL\_REG specified by AVL\_ADDR.

After this step, in order to get the correct sprite character to be printed on the screen, we had to manipulate the DrawX and DrawY values to get which register stores the character that we are trying to print. After we have the 32-bit value from the register, we needed to do some scaling and math (which will be mentioned further in the report) in order to get the correct address that needed to be passed into Font Rom. Once we had this information, the Red, Green, and Blue values were set if the blank signal is not enabled. This was done on every positive edge of the pixel clock signal. Here, we also check the inverse bit to determine if we need to set the foreground color or the background color. The colors were stored in the 600th register in our LOCAL\_REG which was called the CTRL\_REG. The RGB values for foreground or background from this control register were chosen depending on the inverse bit.

#### *Describe at a high level your VGA Text Mode controller IP*

The VGA Text Mode controller IP mainly contained all the elements in our lab 7. This is best described by the diagram below. As seen below, the controller mainly contained Avalon MM slave, the Read/Write byte access logic, VRAM, control register, the data in the Font Rom, our logic for the lab, and the VGA Display. Essentially, the IP provided an environment for the main logic of our entire lab.

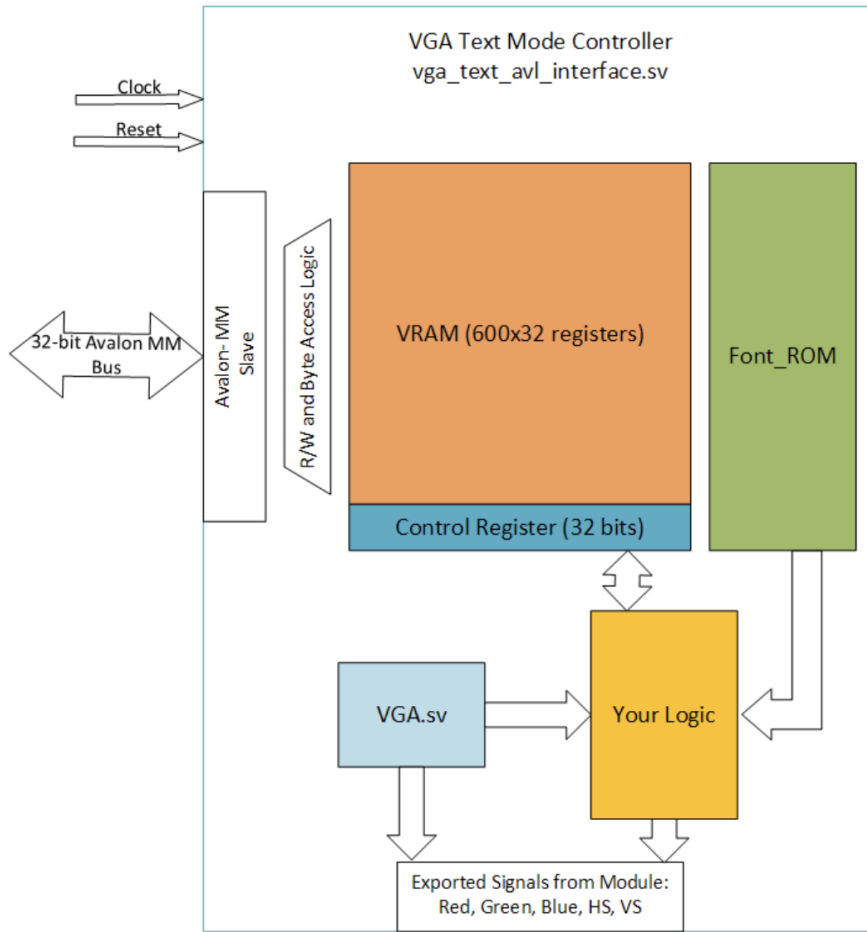


Figure 4. VGA Text Mode Interface Block Diagram

*Describe the logic used to read and write your VGA registers*

The way we read from and wrote to the VGA Registers was mainly by using Avalon-MM Slave Port Interface Signals. There were 7 signals: Read which is High when a read operation is to be performed, Write which is high when a write operation is to be performed, Readdata which is the 32-bit data to be read, Writedata which is the 32-bit data to be written, Address which is the Address of the read or write operation, Byteenable which is the 4-bit active high signal to identify which byte(s) are being written and Chipselect which is High during a read or write operation. As mentioned earlier, when chip select is enabled, by checking the Read signal, the data from the local register specified by the avl\_addr is set to Readdata and by checking the write signal, the data from Writedata is written to the local register specified by the avl\_addr. The writing action was different for a different byte enable value. The different actions as per the byte enable value are shown below.

**Table 2. Byte Enable Description**

| <b>byteenable[3:0]</b> | <b>Write Action</b>        |
|------------------------|----------------------------|
| <b>1111</b>            | Write full 32-bits.        |
| <b>1100</b>            | Write the two upper bytes. |
| <b>0011</b>            | Write the two lower bytes. |
| <b>1000</b>            | Write byte 3 only.         |
| <b>0100</b>            | Write byte 2 only.         |
| <b>0010</b>            | Write byte 1 only.         |
| <b>0001</b>            | Write byte 0 only.         |

All the logic for the reading and writing needed to be done at every positive clock cycle and hence, this was placed in an always\_ff block.

*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).*

To draw the text characters we needed to manipulate our DrawX and DrawY signals to figure out the VGA register, we know that our resolution is 640x480, and since each “word” (defined as each character that we can print out) is 8x16 we can reduce the coordinates to 80x30 since that's the max amount of words we can display. Knowing this we can divide our DrawX and DrawY by 8 and 16 respectively. Instead of dividing we can simply shift Draw X >> 3(drawxsig[9:3]), and DrawY >> 4(drawysig[9:4]) . To index the sprite we want to draw on screen from local registers we did the following calculation:

```

logic [11:0] word;
logic [31:0] word_data;
logic [7:0] char

word = drawysig[9:4] * 80 + drawxsig[9:3];
word_data = LOCAL_REG[word[11:2]];

```

We use the top 10 bits of variable word to access the local registers to pull the encoded data off, then we use the lower two bits (word[1:0]) to determine which of the 4 bytes in word\_data with encoded sprite data we will be drawing to the display. After determining the byte

we store the lower 7 bits of it to a variable called `[7:0] char`, this is the opcode that we will use to access the font rom to print out the character to the screen (the most significant bit will be used to determine the character will be printed out inverted or not). To use the opcode we must multiply it by the row size (16) and add the row offset, our calculation looks like the following:

$$font\_addr = (char[6:0] * 16) + drawysig[3:0]$$

We use the `font_addr` in our instantiation of the `font_rom` module to get `font_data_char` out. To output each pixel in the row correctly we do the calculation:

$$if(font\_data\_char[7 - drawxsig[2:0]] \wedge char[7])$$

This if statement checks the value of the pixel to be printed and XORs it with the invert bit (`char[7]`) if this if statement is true, the output is with normal foreground and background colors. If it is false the output will have inverted colors.

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

From the 8 bits that are extracted for each character, the inverse bit is the 8th bit, which basically suggests if the foreground needed to be colored or the background. If this bit was on or high, we needed to activate the foreground and set the colors accordingly. The background was activated when the 8th bit was low.

| 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 |

Table 3: 7.1 VRAM Sprite encoding

The table above shows how the control register is set up. This register is the essential component in setting up our RGB values according to the inverse bit. When the 8th bit or the inverse bit was high, the foreground was activated which means that the RGB values were set to `FGD_R`, `FGD_G`, and `FGD_B`. When the 8th bit or the inverse bit was low, the background was activated which means that the RGB values were set to `BKG_R`, `BKG_G`, and `BKG_B`.

## Week 2 (Color Text Display)

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

The main modification we made to use on-chip memory-based VRAM was by removing all the registers storing the VRAM and instantiating a RAM instead, which would hold the same memory. We used a dual-port RAM, each had 1 read and 1 write port. This RAM essentially acted as our always\_ff block which would read to and write from local registers in week 1. The RAM, according to the read and write signals, along with the byte enable single, would set the avl\_read and write data to the required value. One port was used for the AVL communication, typically the read and writing of the data, and the other port was to access the VGA data. This means that the second port was only used for reading from the VRAM. Hence, for this port, the read and write were always set to high and low respectively.

*Corresponding modifications to the Platform Designer IP (e.g. Part Editor).*

The only modification that was made to the Platform Designer IP was to change the AVL\_ADDR from 10 bits addressability to 12 bits addressability. Since for week 2 of this lab, each character had a specific color that needed to be chosen from the color palette, the additional information regarding the foreground and background index of the color was stored for each character. Due to this, instead of storing information for 4 characters in one 32-bit value, information for just 2 characters was stored in one 32-bit value. Hence, double the amount of VRAM was required, causing the AVL\_ADDR's addressability to increase from 10 bits to 12 bits.

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

One major difference between Week 1 and Week 2 is the size of our VRAM, in Week 2 our VRAM doubled the size of week 1 at 1200 32 bit registers. Another difference is that each register holds the information of 2 characters and their colors. In Week 2 we are also able to output different foreground colors and background colors for each character. The following calculations are the same from Week 1:

```
logic [11:0] word;  
logic [31:0] word_data;  
logic [15:0] char  
word = drawysig[9:4] * 80 + drawxsig[9:3];
```

Instead of pulling word\_data from our local registers we instead obtain word\_data from our RAM module. Then, we check the least significant bit of the word variable to determine which of the two encoded opcodes we will be outputting and store those 16 bits.

```
word_data = ram_b_out;  
if(word[0]) char = word_data[15:0];  
else word_data[31:16];
```

Following a similar calculation to Week 1 we use char to understand which character from font\_rom we are displaying as well as whether or not we are printing inverted colors or not.

```
font_addr = ((char[14:8] * 16) + drawysig[3:0]);  
if(font_data_char[7 - drawxsig[2:0]] ^ char[15])
```



To display colors we check bits 4-8 (char[7:4]) for normal colors or bits 1-4 ( char[3:0]) for inverted colors. We then check the index of foreground and background colors to check what part of the local color\_palette register we will be accessing. It looks as follows:

```

if(font_data_char[7 - drawxsig[2:0]] ^ char[15])
//if(font_data_char[7 - drawxsig[2:0]] ^ char[7])
begin
//FGD
FGD_IDX = char[7:4];
if(FGD_IDX[0] == 1'b0) //even color number. so right most to access in color palette
begin
red = color_palette[FGD_IDX[3:1]][12:9];
green = color_palette[FGD_IDX[3:1]][8:5];
blue = color_palette[FGD_IDX[3:1]][4:1];
end
else if(FGD_IDX[0] == 1'b1) //odd color number. so left most to access in color palette
begin
red = color_palette[FGD_IDX[3:1]][24:21];
green = color_palette[FGD_IDX[3:1]][20:17];
blue = color_palette[FGD_IDX[3:1]][16:13];
end
end
else
begin
//BGD
BKG_IDX = char[3:0];
if(BKG_IDX[0] == 1'b0) //even color number. so right most to access in color palette
begin
red = color_palette[BKG_IDX[3:1]][12:9];
green = color_palette[BKG_IDX[3:1]][8:5];
blue = color_palette[BKG_IDX[3:1]][4:1];
end
else if(BKG_IDX[0] == 1'b1) //odd color number. so left most to access in color palette
begin
red = color_palette[BKG_IDX[3:1]][24:21];
green = color_palette[BKG_IDX[3:1]][20:17];
blue = color_palette[BKG_IDX[3:1]][16:13];
end
end
end
else
begin
red = 4'h0;
blue = 4'h0;
green = 4'h0;
end
end

```

Figure 5: 7.2 font\_rom data → RGB algorithm

*Additional modifications necessary to support multicolored text.*

As mentioned earlier, we needed to increase the addressability for the AVL\_ADDR from 10 bits to 12 bits in order to store the additional information regarding the colors in the VRAM. Even though most of the information was stored on the on-chip memory, the information regarding the colors was still stored in the FPGA registers. We had a total of 8 FPGA registers which were the color palette registers. Each value was 32 bits, storing the RGB values for 2 colors. The way the memory was set up, if the first bit of the AVL\_ADDR was high, we knew that we are accessing the palette registers. When accessing a specific register from the color palette, we used the bottom 3 bits to index into the 8 palette registers. Along with this, the VRAM at the correct address (for the palette) had to be set up. This was done by creating space

from 0x4B0 - 0x7FF which is unused, and then creating space for the palette from 0x800 - 0x807.

#### *Additional hardware/code to draw paletted colors*

Since the 7.2 required us to create a palette, we had a total of 8 FPGA registers which were the palette registers. In VRAM, these 32-bit registers were stored from addresses 0x800 - 0x807. In order to write to these registers, we had an always\_ff block, similar to week 1, where the registers were written to depending on the byte enable value. Each register stored 2 12-bits worth of information, corresponding to one color in the color palette. As mentioned earlier, we had to create space in the VRAM at the specific addresses, which allowed us to access it easily.

The way the colors were stored in VRAM was by using eclipse. The red green and blue values in the setColorPalette are correctly set in the registers by shifting the colors according to the positions of the RGB values shown in the table below.

| Address | 31-25  | 24-21 | 20-17 | 16-13 | 12-9  | 8-5   | 4-1   | 0      |
|---------|--------|-------|-------|-------|-------|-------|-------|--------|
| 0x800   | UNUSED | C1_R  | C1_G  | C1_B  | C0_R  | C0_G  | C0_B  | UNUSED |
| 0x801   | UNUSED | C3_R  | C3_G  | C3_B  | C2_R  | C2_G  | C2_B  | UNUSED |
| ...     | ...    | ...   | ...   | ...   | ...   | ...   | ...   | ...    |
| 0x807   | UNUSED | C15_R | C15_G | C15_B | C14_R | C14_G | C14_B | UNUSED |

Table 4: 7.2 Color Palette Structure

To find which color each character needs to have, in the 32-bit value stored for each character on the on-chip memory, the foreground and background index is stored, which was a decimal value from 0 to 15. Once we knew if the foreground or background was activated (depending on the inverse bit), the decimal value was used to index into the 8 registers. If the decimal value is an even number, rightmost colors from the registers were accessed. If the decimal value was an odd number, the left-most colors from the registers were accessed. This can be more clearly understood by referring to the table above. Once we knew which color needed to be accessed from the required register, the RGB value was set to the red green and blue value from the register as shown above.

## Block Diagrams

a. Top Level (identical for 7.1 and 7.2)

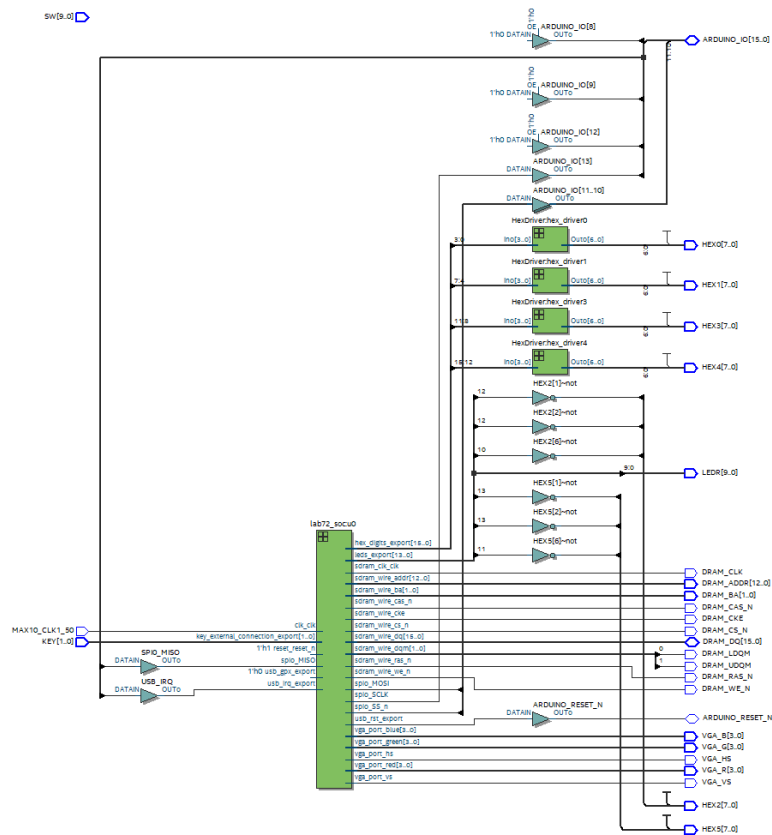
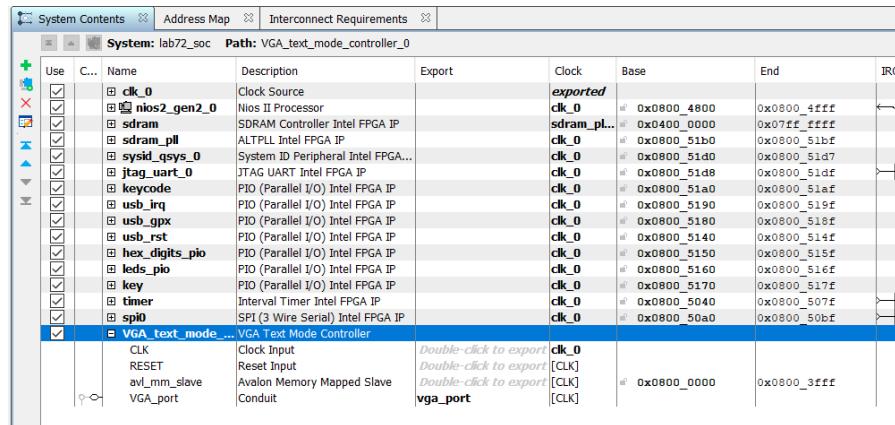


Figure 6: Top Level Block diagram of Lab 7.1 and 7.2

## b. Platform Designer



| Use                                 | C... | Name             | Description                        | Export                 | Clock       | Base        | End         | IRQ |
|-------------------------------------|------|------------------|------------------------------------|------------------------|-------------|-------------|-------------|-----|
| <input checked="" type="checkbox"/> |      | clk_0            | Clock Source                       |                        | exported    |             |             |     |
| <input checked="" type="checkbox"/> |      | nios2_gen2_0     | Nios II Processor                  |                        | clk_0       | 0x0800_4800 | 0x0800_4fff |     |
| <input checked="" type="checkbox"/> |      | sdram            | SDRAM Controller Intel FPGA IP     |                        | sdram_pl... | 0x0400_0000 | 0x07ff_ffff |     |
| <input checked="" type="checkbox"/> |      | sdram_pll        | ALTPLL Intel FPGA IP               |                        | clk_0       | 0x0800_51b0 | 0x0800_51bf |     |
| <input checked="" type="checkbox"/> |      | sysid_qsys_0     | System ID Peripheral Intel FPGA... |                        | clk_0       | 0x0800_51d0 | 0x0800_51d7 |     |
| <input checked="" type="checkbox"/> |      | jtag_uart_0      | JTAG UART Intel FPGA IP            |                        | clk_0       | 0x0800_51d8 | 0x0800_51df |     |
| <input checked="" type="checkbox"/> |      | keycode          | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_51a0 | 0x0800_51af |     |
| <input checked="" type="checkbox"/> |      | usb_irq          | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_5190 | 0x0800_519f |     |
| <input checked="" type="checkbox"/> |      | usb_gpx          | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_5180 | 0x0800_518f |     |
| <input checked="" type="checkbox"/> |      | usb_rst          | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_5140 | 0x0800_514f |     |
| <input checked="" type="checkbox"/> |      | hex_digits_pio   | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_5150 | 0x0800_515f |     |
| <input checked="" type="checkbox"/> |      | leds_pio         | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_5160 | 0x0800_516f |     |
| <input checked="" type="checkbox"/> |      | key              | PIO (Parallel I/O) Intel FPGA IP   |                        | clk_0       | 0x0800_5170 | 0x0800_517f |     |
| <input checked="" type="checkbox"/> |      | timer            | Interval Timer Intel FPGA IP       |                        | clk_0       | 0x0800_5040 | 0x0800_507f |     |
| <input checked="" type="checkbox"/> |      | spi0             | SPI (3 Wire Serial) Intel FPGA IP  |                        | clk_0       | 0x0800_50a0 | 0x0800_50bf |     |
| <input checked="" type="checkbox"/> |      | VGA_text_mode... | VGA Text Mode Controller           |                        |             |             |             |     |
|                                     |      | CLK              | Clock Input                        | Double-click to export | clk_0       |             |             |     |
|                                     |      | RESET            | Reset Input                        | Double-click to export | [CLK]       |             |             |     |
|                                     |      | avl_mm_slave     | Avalon Memory Mapped Slave         | Double-click to export | [CLK]       | 0x0800_0000 | 0x0800_3fff |     |
|                                     |      | VGA_port         | Conduit                            | vga_port               | [CLK]       |             |             |     |

Figure 7: Platform Designer of 7.1 and 7.2

The Platform Designer for Week 1 and Week 2 of Lab 7 is essentially the exact same, the only difference is that the address space of the memory mapped slave increases from 10 - 12 bits. Otherwise it is the same.

Our Platform Designer was adapted from Lab 6 so many modules are unnecessary yet present.

### i. Modules

**clk\_0:** This module is a 50 MHz clock created by the FPGA that provides clock and reset inputs to other modules

**nios2\_gen2\_0:** This module is the 32-bit processor, NIOS II/e, e for the economy, this processor is an optimized version of the full-fledged NIOS II/f, the processor gives out data and instruction as an output to other modules

**onchip\_memory2\_0:** This module is the on-chip memory, it can be used for higher speeds and quicker access to data.

**SDRAM:** This module is the off-chip memory, it is 512 MBits and takes instructions and data from the processor. This module unlike the others gets its clock from the sdram\_pll module which accounts for the difference in phase between the master and slave clocks.

**sdram\_pll:** This module is a clock created for the SDRAM, that has a 1ns delay

**sysid\_qsys\_0:** This module is used to verify that the loaded software on the FPGA is allowed to run on the hardware provided, ensuring that the FPGA is not incompatible with the code.

**jtag\_uart\_0:** This module is used to communicate a serial bit stream between the FPGA and connected computer

**keycode:** This module is a Parallel I/O (PIO) that is used to identify which key on the keyboard has been pressed, it is 8 bits wide

**key:** This module is a 2-bit PIO that is used to represent the two buttons on the FPGA

**usb\_irq, usb\_gpx, usb\_rst:** This module is a 1-bit PIO that is used to connect the keyboard to the FPGA

**hex\_digits\_pio:** This module is a 16-bit PIO that is used to output the keycode on the Hex Display

**leds\_pio:** This module is an 8-bit PIO that is used to output to the leds above the switches on the FPGA

**timer:** This module is used to track time for NIOS II

**spi0:** This module is used to transfer data between the attached peripherals and the FPGA, and controls which peripheral is being read from or written to or if the FPGA is being read from or written to

**VGA\_text\_mode\_controller:** This module is the VGA Text Mode Controller. It is a custom IP, it handles the communication between NIOS and the VRAM. To construct the controller we implemented an Avalon Memory Mapped Slave in hardware, this sets an address, write data, read data, read enable, write enable, byte enable and chip select. For 7.1, there is a read delay of 1 clock cycle. For 7.2, the controller had to map colors to the display according to the VRAM. This is done by setting a VGA port to read the VRAM and get font data from the font rom to then display the pixels with appropriate colors in reference to the VGA\_Controllers.

### Design Resources: 7.1

|               |            |
|---------------|------------|
| LUT           | 32320      |
| DSP           | 0          |
| Memory (BRAM) | 11264      |
| Flip-Flop     | 21793      |
| Frequency     | 123.89 MHz |
| Static Power  | 96.48 mW   |
| Dynamic Power | 0.58 mW    |
| Total Power   | 107.06 mW  |

Table 5 : Design resources and statistics table 7.1

### Design Resources: 7.2

|               |            |
|---------------|------------|
| LUT           | 3627       |
| DSP           | 0          |
| Memory (BRAM) | 142,336    |
| Flip-Flop     | 669        |
| Frequency     | 135.61 MHz |
| Static Power  | 96.85 mW   |
| Dynamic Power | 70.26 mW   |
| Total Power   | 249.48 mW  |

Table 5 : Design resources and statistics table 7.1

### Design Differences

One huge difference between week 1 and week 2 is compilation time. 7.1 uses almost all of the logic elements available on the FPGA to implement all 601 registers without using on chip memory, while 7.2 uses on chip memory to instantiate a true dual port RAM module. One of the benefits of the on chip memory is speed, you can access the data on the RAM much faster than the local registers. However, one tradeoff is your access to the registers. When implemented with

local registers we can manipulate each bit if we wanted to, but, using on chip memory this is not possible since we have to write to the entire register and we have to read the entire register. I believe the design for 7.2 is more efficient, when working on 7.2 we initially re-implemented 7.1 except with on chip memory instead of local registers, and the process of converting from local registers to on chip memory was relatively painless, while taking much less time to compile and debug. Furthermore, in 7.2 our implementation doubles the amount of registers in VRAM being handled, which would be impossible to do with local registers. One last tradeoff that I noticed was the difference in total power. 7.2 uses more than twice as much power as 7.1, however, since our power is in the mW I think this tradeoff is negligible.

## **Conclusion**

Our design was fully functional for both weeks. Both the weeks were relatively simple to understand once we understood the math regarding the DrawX and DrawY values and how these can be manipulated to get the correct sprite data. The main issue we faced in this lab was figuring out how to use the on-chip memory and how to use this to store our VRAM. Once we understood how the RAM worked, this lab was straightforward.

The major extension that we can make to this design that can be useful for our final project would be the drawing of the specific sprites in a specific location on the screen with different colors. For our final project, this is going to be very useful in order to draw the map of our game and the character which can be controlled by our keyboard. The sprites for different components of our final project would be saved in the RAM or the ROM and accessed in a similar way as this lab. Hence, this lab was a good introduction for us to use sprite data and output it onto the screen by setting the right colors.

For this lab, the documentation was pretty detailed, especially with regards to how the VRAM is set up and how the DrawX and DrawY can be manipulated to get the character that needs to be printed on the screen. As for the issues that we faced in this lab, we think the biggest one was the compile time, especially for week 1. The time the code took to compile in the first week was around 8-9 mins even if we made a very small change in our code. I am not sure how this can be improved for the next semester but in our opinion, if the compile times were not as bad, the lab would have been even more straightforward.

Overall, this lab was a really good way to get us familiar with drawing sprites onto the screen by using Font Rom. It also helped us get familiar with on-chip memory again and showed us how it is more efficient in storing the information instead of using FPGA registers. This was again a great lab which will help us a lot in our final project.



## Appendix A

*Module: lab7.sv (identical to lab72.sv)*

Inputs: MAX10\_CLK1\_50, [1:0] KEY, [9:0] SW,

Outputs: [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM\_CLK,  
DRAM\_CKE, DRAM\_LDQM, DRAM\_UDQM, DRAM\_CS\_N, DRAM\_WE\_N,  
DRAM\_CAS\_N, DRAM\_RAS\_N, VGA\_HS, VGA\_VS [3:0] VGA\_R, VGA\_G, VGA\_B,  
[12:0] DRAM\_ADDR, [1:0] DRAM\_BA

Inout: [15:0] DRAM\_DQ, ARDUINO\_IO, ARDUINO\_RESET\_N

Description: This is the top-level module used.

Purpose: The purpose of lab7 is to take input and instantiate modules

*Module: VGA\_controller.sv*

Inputs: Clk, Reset

Outputs: hs, vs, pixel\_clk, blank, sync

[9:0] DrawX, DrawY

Description: Handle vertical and horizontal sync while setting a DrawX and DrawY

Purpose: The purpose of VGA\_controller is to shoot the “electron gun” across the screen and determine the hs and vs to draw pixels accordingly

*Module: hexdriver.sv*

Inputs: [3:0] In0,

Outputs: [6:0] Out0

Description: Takes a nibble or 4 bits and converts it to a 7 bit equivalent that is displayable on the HEX# output LEDs of the FPGA.

Purpose: To allow the seven segment displays to display what the input and output of our code are.

*Module: vga\_text\_avl\_interface.sv*

Inputs: CLK, RESET, AVL\_READ, AVL\_WRITE, AVL\_WRITEDATA, AVL\_CS,  
AVL\_BYTE\_EN, AVL\_ADDR

Outputs: AVL\_READDATA, red, green, blue, hs, vs

Description: This module instantiates the VRAM in on chip memory or local registers for lab 7.2 and 7.1 respectively. In 7.2 this module also instantiates a color palette in local registers. This module also converts the encoded data coming off of VRAM into an address that can be used in font\_rom to print characters onto the display.

Purpose: This module calls VGA\_controller and font\_rom, it also replaces the Color\_Mapper by passing out RGB values ready to be displayed. It also acts as a read/write interface between the CPU and the VRAM.