ECE 385

Spring 2023

Lab 7

Experiment #7

Aleena Majeed & Ali Chaudry
aleenam2 & achau9
17:50 - 17:55

1. Introduction

a. Briefly summarize the operation of the VGA interface, what are we trying to accomplish with this design?
b. You should address how the design you created builds on top of the basic one provided for Lab 6.2.

2. Written Description of Lab 7 System

a. Week 1 (Monochrome Text Display)
i. Written Description of the entire Lab 7 system
ii. Describe at a high level your VGA Text Mode controller IP
iii. Describe the logic used to read and write your VGA registers
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.
b. 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:

1. Modification of register-based VRAM to on-chip memory-based VRAM. How did your design share the limited on-chip memory ports?
7.4
2. Corresponding modifications to the Platform Designer IP (e.g. Part Editor).
3. Modified sprite drawing algorithm with the updated indexing equations from on-screen pixels to VRAM.
4. Additional modifications necessary to support multicolored text.
5. Additional hardware/code to draw paletted colors

3. Block Diagram

a. This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module.
b. Note that depending on your layout of the registers inside your main module, the Quartus view may be illegible, in which case you should draw a block diagram using software. You may start from the provided materials (e.g. in IAMM), but you should fill in the specific signals between the modules and the inside subcomponents within each module.
c. 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 VGA controller component.
d. If your design has a state machine, you should include a State Diagram as well.

4. Module Descriptions

a. A guide on how to do this was shown in the Lab 6 report outline. Do not forget to describe the Platform Designer generated file for your Nios II system! When describing the generated file, you should describe the PIO blocks added beyond those just needed to make the NIOS system run (i.e. the ones needed to communicate with the USB chip and other components). The Platform Designer view of the Nios II system is helpful here.

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. Which design is more efficient, what are the tradeoffs?

6. Conclusion

a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.
b. What are some potential extensions of this design, what did you learn in this lab that might be useful for your Final Project?
c. 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.

1. Introduction

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

A VGA is a video graphics array which the interface uses to connect a computer to some sort of display, in our case a monitor, and the interface displays the data to the screen as a matrix of pixels. The VGA does this using an electron beam that draws the pixels from right to left and top to bottom. In our lab 7, we used the VGA to draw characters that were stored in a ROM. We did this by accessing the address of the character (since this lab is data driven), the location of the character on the 640x480 matrix (dimensions of the monitor screen in the unit of pixels), and accessing the correct color palette for each character that we wanted to draw. In this lab, the main way that we were able to use the VGA was to display the colors of the sprites and the backgrounds. We did this by having a dedicated control register that would hold the colors that we want to print.

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

The design we created for Lab 7 builds on top of the basic one we were provided for in Lab 6.2 in the sense that we're using the VGA interface to display certain text/characters and their respective colors, and having this information stored in either registers or on-chip memory. In lab 6.2, we utilized the NIOs II and the VGA interface to display a ball on the monitor with certain colors, and then used keyboard inputs to make the ball move in a certain direction with respect to the pressed keycode. We use the NIOs II processor in this lab to read and write data into the registers which would contain all of the information needed to move the ball according to the input we provide using the SPI (Serial Port Interface). In lab 7, we built off of the simplicity from lab 6.2 by once again using the VGA interface, but this time we instead were writing text to the screen, and the text had specific colors that correspond to what we were writing. In this lab, we use the Avalon MM to send signals to how we want our program to operate on a certain register, for week 1, and for week 2 we used this to send the correct signals for how we want our program to operate using on-chip memory.

2. Written Description of Lab 7 System

a. Week 1 (Monochrome Text Display)

i. Written Description of the entire Lab 7 system

Lab 7 was done over the span of two weeks. For week 1, we created a monochrome graphics controller in order to connect the NIOs II to the Avalon bus. In this week of the lab, we used 601 registers to implement this: 600 registers containing VRAM words (glyphs), and then 1 register (601) as a control register that will control the color that we print out using the VGA interface. We used a fixed function in order to implement this monochrome graphics controller because it doesn't require the use of on-chip memory (which we used in week 2). Since there's a total of 2400 characters that can be displayed, our screen is divided into a 30 x 80 matrix, where there are 30 rows and 80 columns. Within each block of this matrix, it contains a glyph represented by 8 x 16 pixels. We are able to draw these pixels by accessing the Avalon bus to access one of our 600 registers, and each register holds 4 different characters that we can draw to the screen. These 600 registers are 32-bits wide, containing 4 glyphs and an invert bit for each that determines if we want to invert the color of the glyph being drawn. In the single control register, this is also 32-bits wide and contains the colors for the foreground and background of each glyph.

ii. Describe at a high level your VGA Text Mode controller IP

The VGA Text Mode controller IP used in our Lab 7.1 is used to connect the signals between the Avalon MM and the VGA display. This module is responsible for handling the reading and writing of the registers, which hold the colors and characters that we displayed using the VGA for this lab. 600 of these 8-bit registers hold 4 characters each: with bytes dedicated as an inverse bit to draw the glyph in different colors and bits to just draw the glyph itself, and 1 control register of 8-bits to define the foregrounds and background colors, as well as the inverse glyph. Also, we use this module to instantiate the font_rom and vga_controller modules, and to implement the Avalon-MM interface. The "font_rom" module is the module that contains the data (characters) in the ROM in row major order, which we access in this module in order to write these characters onto the VGA display.

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

In order to read and write into the VGA registers for Lab 7.1, we do this in the vga_text_mode_controller module. We have a local variable we call "LOCAL_REG", which is a 32-bit array that exists for our 601 registers. We do the writing/reading logic inside of an always_ff block, as writing and reading in this lab require synchronous logic, and we first have a condition that if the RESET button on the FPGA is pressed, that we want to clear all 601 of our registers. In other words, we are making sure that the data from the VRAM contained in each register is a value of 0. We then check that our chip select signal is high, as we can only read and write when this signal is high, and then check to see if our AVL_WRITE signal is high or if our AVL_READ signal is high to see which we want to execute. If our AVL_WRITE is high, then we have a series of if statements checking our 4-bit byte enable signal. We then check the

following byte enable combinations to implement the corresponding writes, which will write the proper bits of AVL_WRITEDATA into the propper bits of the LOCAL_REG:

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

Corresponding write action to each byte enable signal

However, when the AVL_READ bit is higher than the AVL_WRITE bit, then we want to read a value from our LOCAL_REG rather than writing a value into it. So to do this, we check that the AVL_READ signal is high, and then we put the value of the LOCAL_REG at the specific address into the AVL_READDATA: AVL_READDATA[31:0] <= LOCAL_REG[AVL_ADDR][31:0].

```
always_ff @(posedge CLK) begin


		if(RESET)begin
			for(int i = 0; i<601; i++)
				LOCAL_REG[i][31:0] <= 32'h00000000;
			end


		else if(AVL_CS)

		begin


		if((AVL_WRITE) && (~AVL_READ))
			begin


//logic inv_bit

	if(AVL_BYTE_EN == 4'b1111)
		LOCAL_REG[AVL_ADDR][31:0] <= AVL_WRITEDATA[31:0];

	else if(AVL_BYTE_EN == 4'b1100)
		LOCAL_REG[AVL_ADDR][31:16] <= AVL_WRITEDATA[31:16];

	else if(AVL_BYTE_EN == 4'b0011)
		LOCAL_REG[AVL_ADDR][15:0] <= AVL_WRITEDATA[15:0];

	else if(AVL_BYTE_EN == 4'b1000)
		LOCAL_REG[AVL_ADDR][31:24] <= AVL_WRITEDATA[31:24];

	else if(AVL_BYTE_EN == 4'b0100)
		LOCAL_REG[AVL_ADDR][23:16] <= AVL_WRITEDATA[23:16];

	else if(AVL_BYTE_EN == 4'b0010)
		LOCAL_REG[AVL_ADDR][15:8] <= AVL_WRITEDATA[15:8];

	else if(AVL_BYTE_EN == 4'b0001)
		LOCAL_REG[AVL_ADDR][7:0] <= AVL_WRITEDATA[7:0];
end


		else if((AVL_READ) && (~AVL_WRITE))
		AVL_READDATA[31:0] <= LOCAL_REG[AVL_ADDR][31:0];



	end


	end
```

Screenshot of SV code displaying the read and write logic for week 1

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).

To draw the text characters to the screen, we did a series of arithmetic operations in order to find out certain information about our glyph: where we were going to draw the glyph in the 8x16 matrix, what character the register is holding that we will draw, and then if the character will be drawn with an inverted color. To start, we use the variables DrawX and DrawY that we receive from the VGA_controller module.

The DrawX and DrawY signals pertain to the top leftmost corner of the text character that we are drawing. And since each glyph is represented by 8 x 16 pixels in our ROM, we use these dimension to find the exact row and column that the glyph is drawn on in the larger 80x30 matrix

that encapsulates the entire monitor display (640 / 8 = 80 and 480 / 16 = 30). To do this, divide our DrawX by 8 and our DrawY by 16 to obtain the correct number of pixels that our glyph is drawn in our VRAM.

We then needed to calculate the byte address and we do this using the row major order equation which is the (number of rows * number of columns + the current column). In this case, our equation to find the byte address looks like (80 * DrawY/16 + DrawX/8). We use this equation because of the fact that the data for the glyphs is located in row major order in the ROM, and we are only able to access data through one row at a time. Finally, to obtain the word address to get the correct index for the VRAM, we must multiply the byte address that we just obtained by 4 because this accounts for the number of characters that are stored in each register that must be chosen from to draw to the screen.

```
drawXNew= drawX[9:3]; // divided by 8
drawYNew = drawY[9:4]; //divide by 16
byteAddress = (80*drawYNew)+drawXNew; // (80*DrawyNew)+DrawxNew //lower two bits inidcate the dedicated bytes
word_address = byteAddress[31:2];  // word_address/4
```

Arithmetic used to draw 4 characters from VRAM and ROM

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

The inverse bit was needed to implement reverse video functionality. The inverse was used to invert the colors of the text and background. For example it was used to help make the text appear white on the black background. If the Inversion bit has a value of 1, the final pixel value is obtained by inverting the 1 or 0 value from the font ROM. This value is then utilized to determine whether the foreground (1) or background (0) color is to be drawn. The red, green, and blue signals are linked to the respective foreground and background signals in the control register, and the monochrome characters are drawn on the screen based on the final pixel value.

The control register stores both the background and foreground colors, which are interchanged when the inverse bit is set to a high state. By connecting the red, green, and blue signals to their respective background and foreground counterparts in the control register and selecting them based on the resulting pixel value, the monochrome characters are rendered on the screen.

Table 4. Bit Encoding for VRAM (Word Addresses 0x000-0x257)

Bit	31	30-24	23	22-16	15	14-8	7	6-0
Function	IV3	CODE3	IV2	CODE2	IV1	CODE1	IV0	CODE0

IVn = Inverse bit N

CODEn = Glyph code from IBM Codepage 437

Bit encoding for the VRAM of 4 characters in 32-bits

Table 6. Bit Encoding Control Register (Word Address 0x258)

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

BKG_R/G/B = Background color, flipped with foreground when IVn bit is set
FGD_R/G/B = Foreground color, flipped with background when IVn bit is set

Bit encoding for the VRAM of 2 characters in 32-bit

b. 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:

To support multi-color text, several hardware changes were necessary. The existing design would have only supported a limited number of colors, so new registers had to be added to store additional colors for multi-color text. To accomplish this, C code was written to implement 16 color registers, with each register holding 2 characters - one for foreground and the other for background color.

Since the existing RAM IP would not have been sufficient to accommodate the additional registers, on-chip memory was used for modifying the register-based VRAM. This change helped to reduce the cost and complexity of the design, making it more efficient and easier to implement.

In addition to the changes mentioned above, the existing logic was also modified to correctly index the palettes since each of the 16 registers now contained two colors. The LSB of the foreground and background index determined the position of each color in the register, and values were read by the VGA cable.

Overall, these changes were necessary to create a design that could display a wider range of colors for text and other graphics. By adding more color registers and modifying the logic to support them, the design became more flexible and capable of displaying complex graphics with multiple colors. Using on-chip memory also helped to improve the efficiency and reduce the complexity of the design, making it more practical for real-world applications.

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

For week 2 of this lab, we utilized on-chip memory for our glyph registers and our color palettes, as this gives us more memory space to work with rather than just using the FPGA's registers. This change was necessary because we were changing the monochrome graphics display to a supported color text, where each character we print has a specific color that it needs to print in and therefore, we had to extend each character to 2 bytes. In order to do this, we instanted a RAM in our vga_text_mode_controller module. By adding the RAM to this module, we no longer needed to write character data into 600

registers, because we are now using the RAM to hold this data. Instead, we are only using registers for our color palette, which we can still read and write from just as we did in week 1 of the lab.

As stated above, we moved the data of glyphs onto on-chip memory, but we are keeping the color palettes in registers. On the contrary from week one, each address now holds 2 characters, there being 2 bytes per character. Byte 1 is the same as week 1, it contains the bits for the glyph and its inverse bit. But byte 2 contains 2, 4-bit colors and this information is the index to the palette register that contains all the supporting colors for this week of the lab.

```
drawXNew= drawX[9:3]; // divided by 8
drawYNew = drawY[9:4]; //divide by 16
byteAddress = (80*drawYNew)+drawXNew; // (80*DrawyNew)+DrawxNew //lower two bits inidcate the dedicated bytes
word_address = byteAddress[31:1];  // word_address/2
```

Arithmetic used to draw 4 characters from VRAM and ROM

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

In regards to the Platform Designer IP for week 2 of this lab, we changed very little in the platform designer. We keep the same signals of the read, write, readdata, writedata, address, byteenable and chipselect. However, we needed to change the width of the address from 10 bits to now being 12 bits. This increase in bits was necessary to accommodate for the additional VRAM and the now color palette being used in the lab, which can be accessed through a single bit of the address: ADDR[11] to select between the color palette and the VRAM.

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

As stated previously, our 32-bit address now only holds 2 characters instead of 4, as we need to account for each character having a respective foreground and background color that needs to be displayed. To account for this in the calculations for finding our byte and word address, rather than multiplying our byte address by 4, we instead multiply it by 2.

So, to draw the text characters to the screen, we did a series of arithmetic operations in order to find out certain information about our glyph: where we were going to draw the glyph in the 8x16 matrix, what character the register is holding that we will draw, and then if the character will be drawn with an inverted color. To start, we use the variables DrawX and DrawY that we receive from the VGA_controller module.

The DrawX and DrawY signals pertain to the top leftmost corner of the text character that we are drawing. And since each glyph is represented by 8 x 16 pixels in our ROM, we use these dimension to find the exact row and column that the glyph is drawn on in the larger 80x30 matrix that encapsulates the entire monitor display (640 / 8 = 80 and 480 / 16 = 30). To do this, divide our DrawX by 8 and our DrawY by 16 to obtain the correct number of pixels that our glyph is drawn in our VRAM.

We then needed to calculate the byte address and we do this using the row major order equation which is the (number of rows * number of columns + the current column). In this case, our equation to find the byte address looks like (80 * DrawY/16 + DrawX/8). We use this equation because of the fact that the data for the glyphs is located in row major order in the ROM, and we are only able to access data through one row at a time. Finally, to obtain the word address to get the correct index for the VRAM, we must multiply the byte address that we just obtained by 2 because this accounts for the number of characters that are stored in each 32 address that must be chosen from to draw to the screen.

4. Additional modifications necessary to support multicolored text.

Additional modifications that were necessary to support the multi-colored text was creating color palette registers that can hold 2 colors per register. This means that we had to create modifications to choose between 2 different foreground and background colors as well as which color to draw the specified glyph in. All of these additions will be described in the next section.

Table 7. Bit Encoding for VRAM (Color Mode, Word Addresses 0x000-0x4AF)

Bit	31	30-24	23-20	19-16	15	14-8	7-4	3-0
Function	IV1	CODE1	FGD_IDX1	BKG_IDX1	IV0	CODE0	FGD_IDX0	BKG_IDX0

Note the additional attributes, where:

FGD_IDXn is the **Foreground Color Index for character n**
BKG_IDXn is the **Background Color Index for character n**

Bit encoding for the VRAM of 2 characters in 32-bits

Table 9. Color Palette Organization (0x800-0x807)

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

Bit distribution for color palette 8 color palette registers

5. Additional hardware/code to draw paletted colors

```
always_ff @(posedge CLK) begin

      if(AVL_ADDR[11] & AVL_WRITE & AVL_CS ) //if this bit is 1 then we want to read and write to the palette registers
      color_palette_register[AVL_ADDR[2:0]] <= AVL_WRITEDATA;

      else if(AVL_ADDR[11] & AVL_READ & AVL_CS)
      AVL_READDATA <= color_palette_register[AVL_ADDR[2:0]];

      else if (~AVL_ADDR[11] & AVL_READ & AVL_CS)
      AVL_READDATA <= dummy;


   end
```

Modified SV code to read and write into registers for 7.2

For this lab, in order to support multicolored text, we had to have dedicated color palette registers, which we used from the FPGA, to hold the 16 different colors that could be used on our text display. The logic to read and write with our palette register is the same as from lab 7.1, where we check if our AVL_CS and AVL_READ/WRITE signals are high, but for lab 7.2, we need to also check that we are accessing the color palette to read or write from. To do this, we also must check that we are accessing the most significant bit of the address, AVL_ADDR[11].

```
   always_ff @(posedge pixelclk) begin
//////////////////////////////////////////////////////////////////////////////////////////////////////////
      if(blank)

   begin

   if(code[15] ^ data[7-drawX[2:0]] )   //you wanna draw foreground
            begin  //then draw fg0
               if(FGD_IDX[0] == 1'b0)
                  begin
                     red <= color_palette_register[FGD_IDX[3:1]][12:9] ; //changed to 7:4 from 3:1
                     blue <= color_palette_register[FGD_IDX[3:1]][4:1];
                     green <= color_palette_register[FGD_IDX[3:1]][8:5];
                  end
               else if (FGD_IDX[0] == 1'b1)
                  begin
                     red <= color_palette_register[FGD_IDX[3:1]][24:21] ;
                     blue <= color_palette_register[FGD_IDX[3:1]][16:13];
                     green <= color_palette_register[FGD_IDX[3:1]][20:17];
                  end
   end

   else
   begin
            if(BKG_IDX[0] == 1'b0)
            begin
               red <= color_palette_register[BKG_IDX[3:1]][12:9] ;
               blue <= color_palette_register[BKG_IDX[3:1]][4:1];
               green <= color_palette_register[BKG_IDX[3:1]][8:5];
            end
            else if(BKG_IDX[0] == 1'b1)
            begin
               red <= color_palette_register[BKG_IDX[3:1]][24:21] ;
               blue <= color_palette_register[BKG_IDX[3:1]][16:13];
               green <= color_palette_register[BKG_IDX[3:1]][20:17];
            end

   end
```

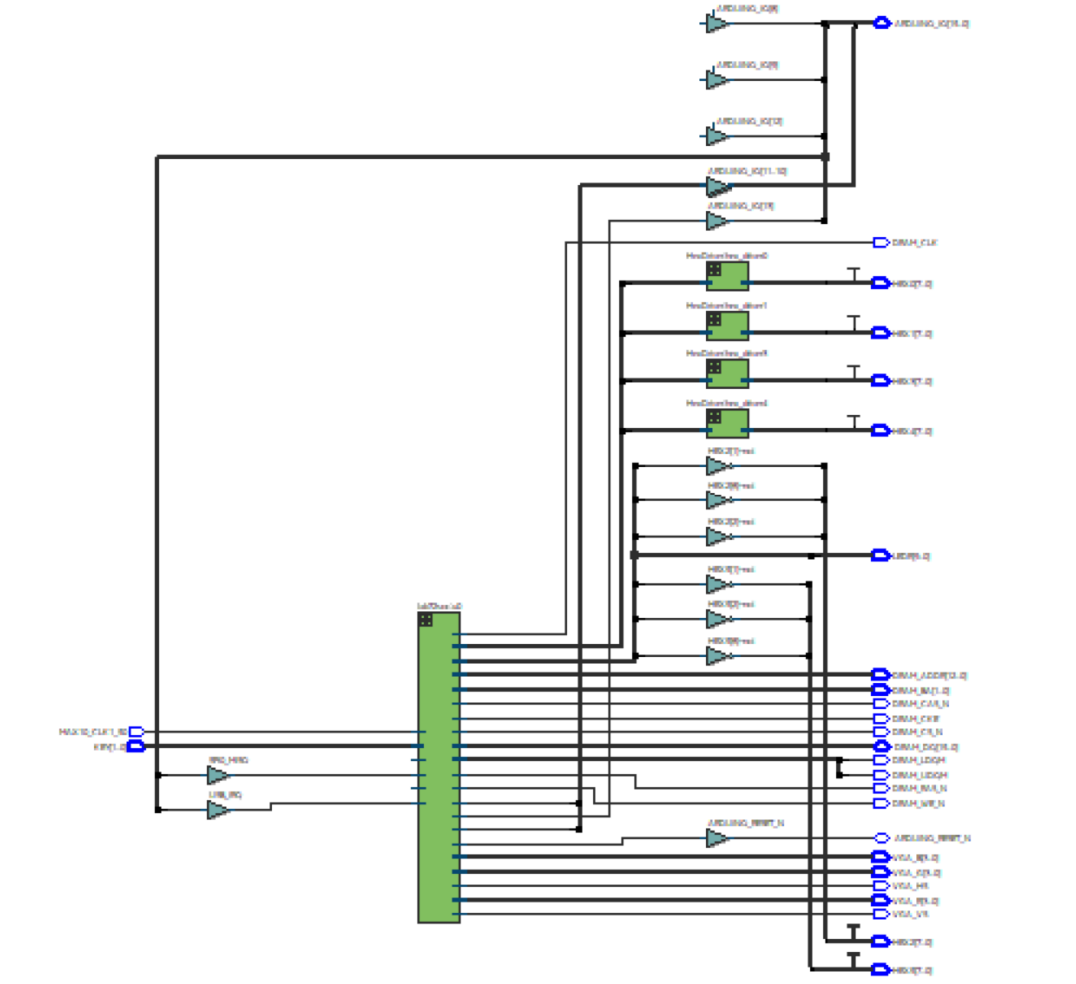
Modified SV code to assign the red, blue and green from the color palette

In lab 7.2, the red, blue, and green colors for the foreground and background will change depending on which character we want to draw. IWe first check if we even want to draw the foreground or the background colors, and then check to see if we want to draw the foreground or background color of the first or second character. If we want to draw the

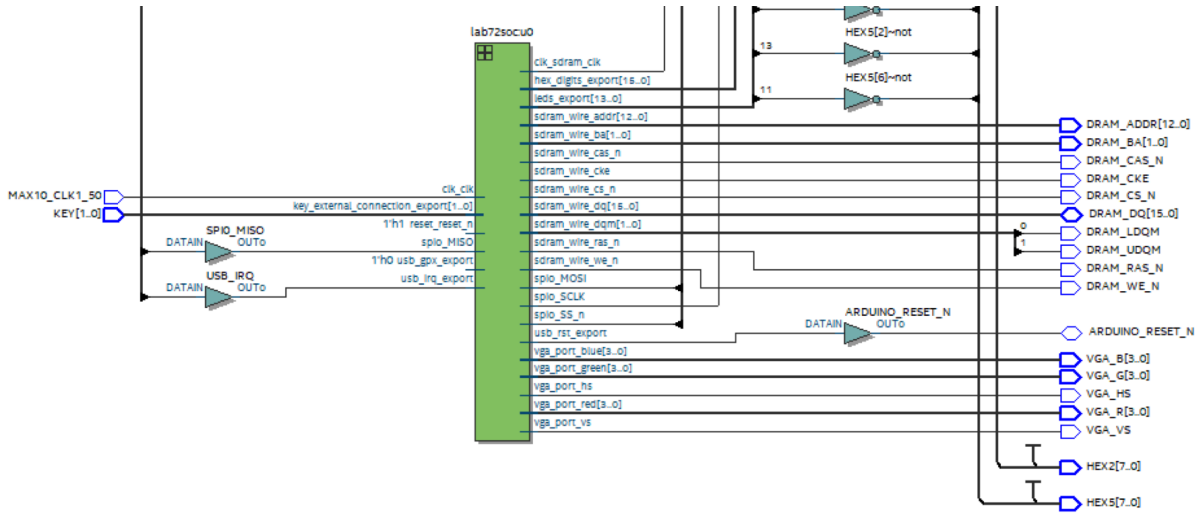
character indexed at 0, then we will write the value of the upper bits of the foreground from the color palette or the lower bits of the background from the color palette into the red, blue and green variables that are to be displayed on the monitor. If we want to draw the character indexed at 1, then we give the red, green and blue values either the upper bits of the foreground or the lower bits of the background.

3. Block Diagram

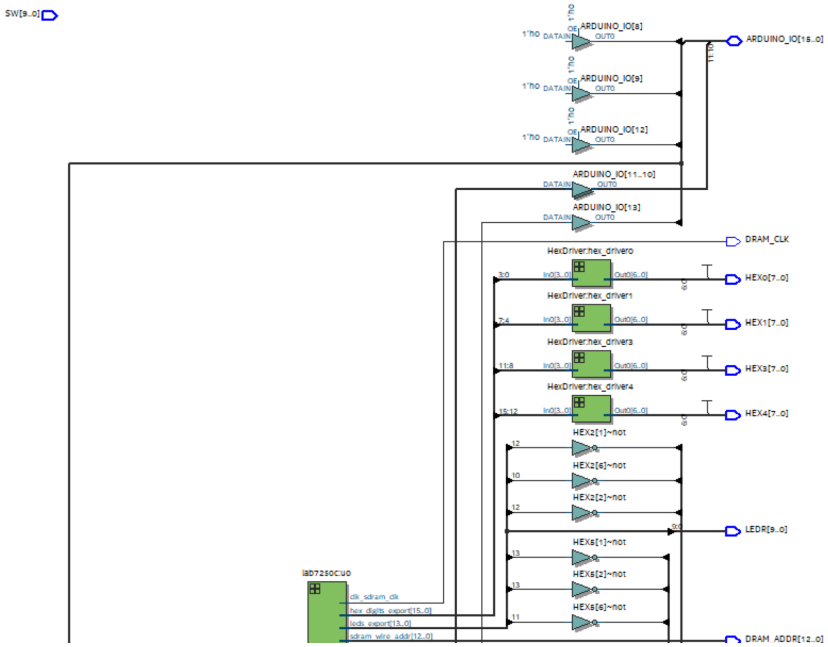
a. This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module.
b. Note that depending on your layout of the registers inside your main module, the Quartus view may be illegible, in which case you should draw a block diagram using software. You may start from the provided materials (e.g. in IAMM), but you should fill in the specific signals between the modules and the inside subcomponents within each module.
c. 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 VGA controller component.
d. If your design has a state machine, you should include a State Diagram as well.



LAB 7 RTL BLOCK DIAGRAM



Zoomed in screenshot of top level



Zoomed in screenshot of the rest of the modules in lab7

4. Module Descriptions

a. A guide on how to do this was shown in the Lab 6 report outline. Do not forget to describe the Platform Designer generated file for your Nios II system! When describing the generated file, you should describe the PIO blocks added beyond those just needed to make the NIOS system run (i.e. the ones needed to communicate with the USB chip and other components). The Platform Designer view of the Nios II system is helpful here.

- **Vga_text_avl_interface.sv:**
  - **module vga_text_avl_interface (**
  - **input logic CLK,**
  - **input logic RESET,**
  - **input logic AVL_READ,**
  - **input logic AVL_WRITE,**
  - **input logic AVL_CS,**
  - **input logic [3:0] AVL_BYTE_EN,**
  - **input logic [11:0] AVL_ADDR,**
  - **input logic [31:0] AVL_WRITEDATA,**
  - **output logic [31:0] AVL_READDATA,**
  - **output logic [3:0] red, green, blue,**
  - **output logic hs, vs**
  - **);**

  Function: This module was responsible for instantiating the vga_controller module and font_rom module. The instantiations and top level module connects the hardware components with the software and allows the keyboard peripherals to interact with the monitor peripherals.

- **VGA_controller.sv:**
  - **module vga_controller (**
  - **input Clk**
  - **Reset,**
  - **output logic hs,**
  - **pixel_clk,**
  - **blank,**
  - **sync,**
  - **output [9:0] DrawX,**
  - **DrawY**
  - **);**

  Function: FUNCTION: Every screen consists of an electron gun that traverses the display to fill in individual pixels with color. The vga_controller module is responsible for managing the

vertical and horizontal synchronization, as well as the electron gun's movements, to enable the accurate drawing of these pixels.

- **lab7.sv(toplevel):**
  - **module lab7 (**
  - **input MAX10_CLK1_50,**
  - **input [1: 0] KEY,**
  - **input [9: 0] SW,**
  - **output [9: 0] LEDR,**
  - **output [7: 0] HEX0,**
  - **output [7: 0] HEX1,**
  - **output [7: 0] HEX2,**
  - **output [7: 0] HEX3,**
  - **output [7: 0] HEX4,**
  - **output [7: 0] HEX5,**
  - **output DRAM_CLK,**
  - **output DRAM_CKE,**
  - **output [12: 0] DRAM_ADDR,**
  - **output [1: 0] DRAM_BA,**
  - **inout [15: 0] DRAM_DQ,**
  - **output DRAM_LDQM,**
  - **output DRAM_UDQM,**
  - **output DRAM_CS_N,**
  - **output DRAM_WE_N,**
  - **output DRAM_CAS_N,**
  - **output DRAM_RAS_N,**
  - **output VGA_HS,**
  - **output VGA_VS,**
  - **output [3: 0] VGA_R,**
  - **output [3: 0] VGA_G,**
  - **output [3: 0] VGA_B,**
  - **inout [15: 0] ARDUINO_IO,**
  - **inout ARDUINO_RESET_N**
  - **);**

Function: This module is the top level module of our Lab 7. It includes various inputs and instantiates the lab72soc module and is responsible for assigning the HEX drivers to make them outputs on the FPGA. It is also responsible for Aruduino wire instantiations of SPI ports like the SPI clock, MISO, MOSI, etc.

- **font_rom.sv:**
  - **module font_rom (**
  - **input [10:0] addr,**
  - **output [7:0] data**
  - **);**

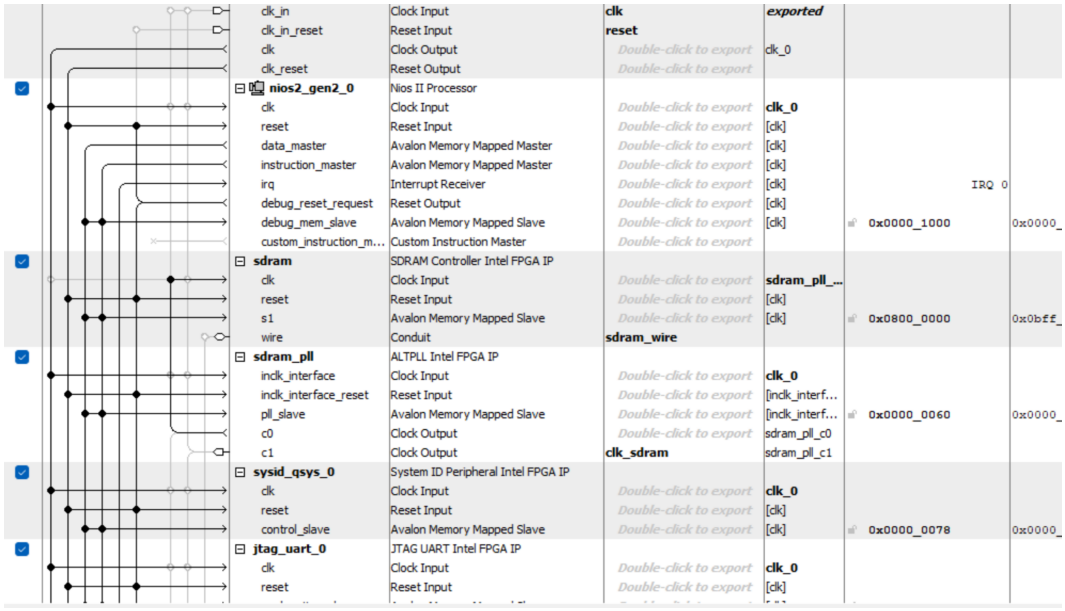
Function: The module implements the asynchronous ROM by making 8-bit rows containing the pixel data that will be used to draw the glyphs. The pixel data contains 0`s and 1's to indicate the drawing of the foreground and background colors.

- **hex driver.sv:**
  - **module HexDriver (**
  - **input [3:0] In0,**
  - **output logic [6:0] Out0**
  - **);**

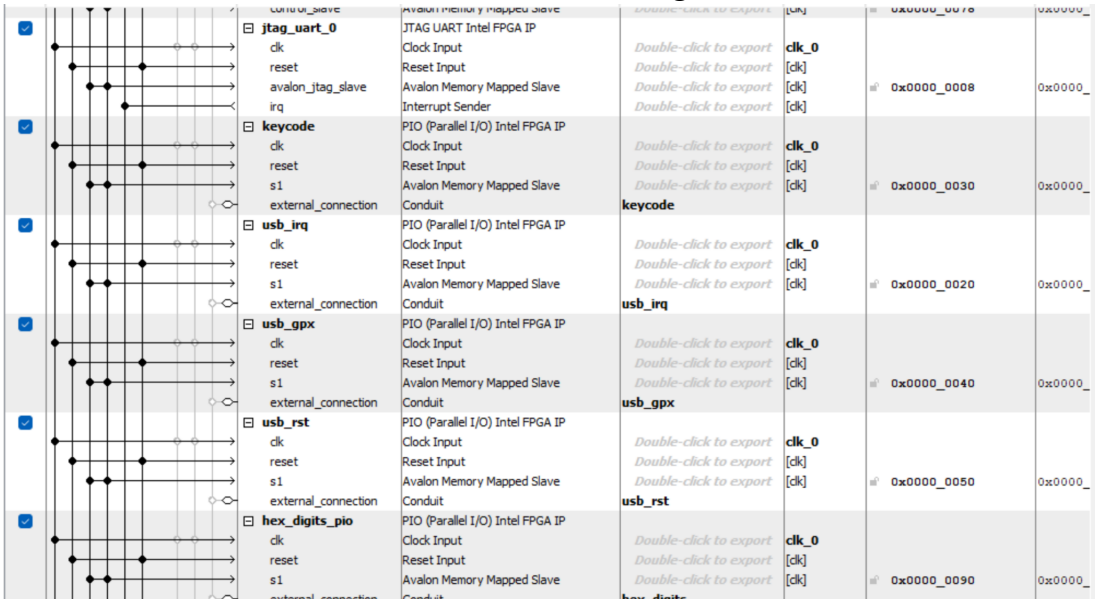
Function: This module simply contains if statements to apply the correct input/output logic so that the hex drivers of the FPGA get the correct values depending on which LED values are high.

- **Ram.v (only for 7.2):**
  - **module ram (**
  - **address_a,**
  - **address_b,**
  - **byteena_a,**
  - **clock,**
  - **data_a,**
  - **data_b,**
  - **rden_a,**
  - **rden_b,**
  - **wren_a,**
  - **wren_b,**
  - **q_a,**
  - **q_b**
  - **);**

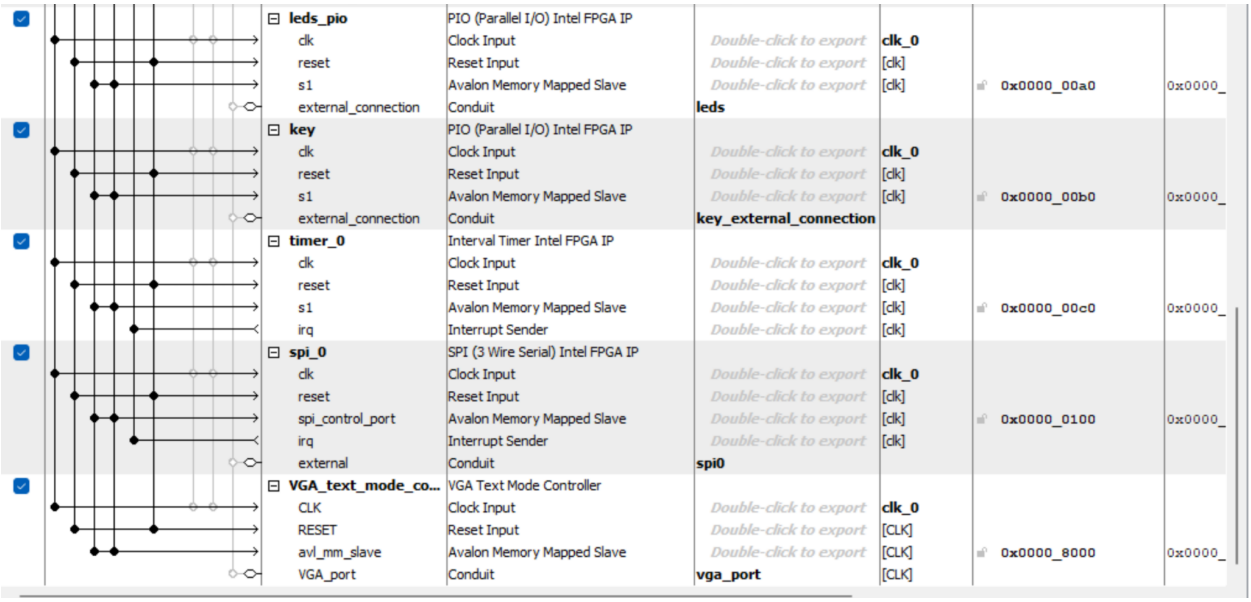
Function: This module instantiates the on chip memory for the FPGA used in this lab. It contains the signals needed to read and write to memory.



Lab 7.1 PIO blocks image 1



Lab 7.1 PIO blocks image 2



Lab 7.1 PIO blocks image 3

- **clk_0 -** This is a 50 MHz clock from the FPGA and includes a clock and a reset which are used to clock and reset other modules.

- **nios2_gen2_0 -** One of the key advantages of the NIOS II processor is its ease of use. It is designed to be programmed in C/C++, and its software development tools are integrated with widely used development environments like Eclipse. This makes it easy for software developers to write and debug code for the processor, reducing the time and effort required for development.

- **sdram -** In Nios II, the SDRAM is typically connected to the Nios II processor through a memory controller, which manages the transfer of data between the processor and the SDRAM. The memory controller provides an interface between the processor and the memory module, and handles tasks such as initializing the memory, issuing read and write commands, and managing the refresh cycles. When the Nios II processor needs to access data from the SDRAM, it sends a read request to the memory controller. The memory controller retrieves the requested data from the SDRAM and sends it back to the processor. Similarly, when the processor needs to write data to the SDRAM, it sends a write request to the memory controller, which writes the data to the appropriate memory location. The SDRAM's speed and capacity can affect the overall performance of a Nios II system. To optimize performance, the memory controller must be configured to work efficiently with the specific SDRAM device used in the system. This can involve setting timing parameters, configuring the refresh rate, and configuring other memory controller settings. Overall, the use of SDRAM in a Nios II system allows for efficient data storage and retrieval, enabling the system to run faster and more efficiently.

- **sdram_pll -** The SDRAM clock is linked to a clock signal that is intentionally delayed by 1 ns. This delay is implemented to compensate for any phase discrepancies that may arise between the master and slave clocks.

- **sysid_qsys_0 -** This is used to check for errors in the connections from the hardware to software components that are connected. The hardware and software must be updated to the latest changes so they are compatible. This is why steps such as generating BSP are necessary.

- **jtag_uart__0 -** The JTAG UART is a hardware module that provides a communication interface between the FPGA and the computer or another device. The JTAG UART module allows for bi-directional communication between the device and an external device or computer. It provides a standard serial interface, which allows for the transfer of data in a standardized format. This interface is typically used for debugging and diagnostic purposes, as well as for firmware updates.

- **keycode -** The Keycode refers to an essential 8-bit PIO input that enables the processor to retrieve the specific code corresponding to a pressed key on a USB keyboard.

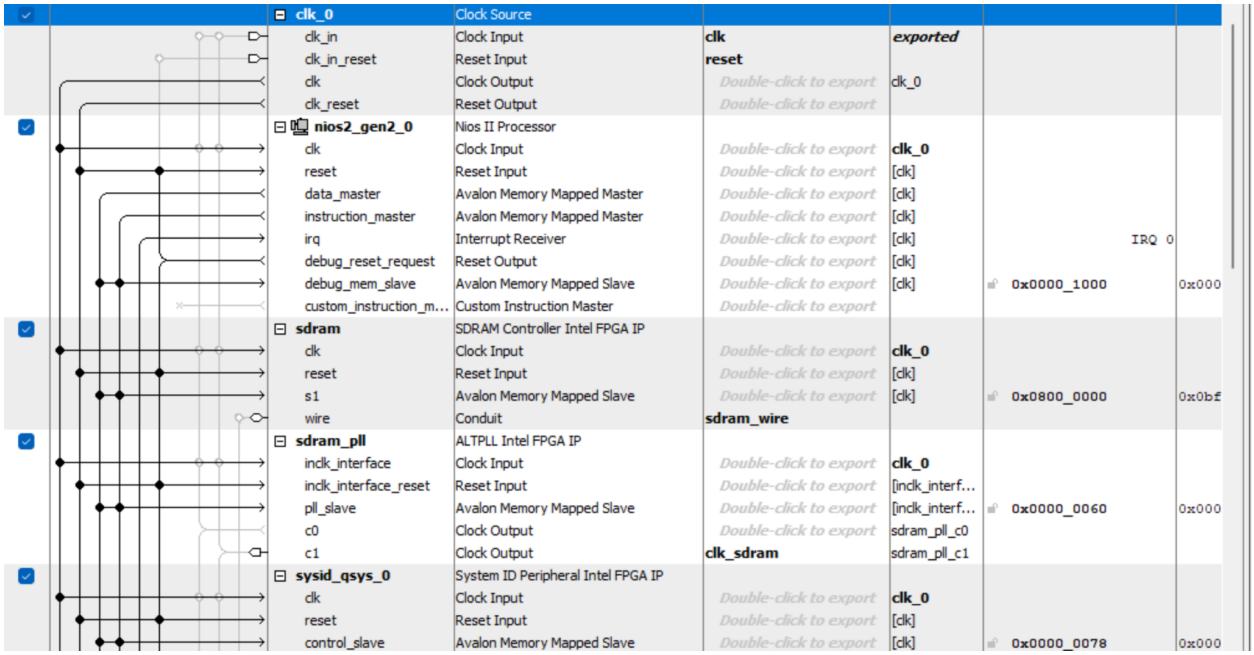
- **usb_irq -** Refers to the interrupt signal used by the USB device to notify the host (computer) about an event that requires attention, such as the insertion of a new device or the completion of a data transfer. This is needed to utilize the USB keyboard with the FPGA.
- **usb_gpx -** Stands for USB General Purpose Input/Output. It provides a set of pins that can be configured by the device designer for various purposes, such as controlling LEDs, reading switches, or interfacing with sensors. This is needed to utilize the USB keyboard with the FPGA.

- **usb_rst** - A signal used to reset the USB device or put it in a low-power mode. It is typically activated by the host, either through a software command or a hardware pin. This is needed to utilize the USB keyboard with the FPGA.

- **hex_digits_pio -** The term hex_digits_pio refers to a parallel input/output interface that can communicate hexadecimal data between a microcontroller or processor and an external device. In the context of the DE-10 board, this interface can be used to output the keycode of the currently pressed key on the hex display. By utilizing a 16-bit parallel interface, the transfer of multiple digits can be accomplished simultaneously, resulting in faster and more efficient communication.

- **key -** This was used in lab 6 - not needed for Lab 7. The Keycode refers to an essential 8-bit PIO input that enables the processor to retrieve the specific code corresponding to a pressed key on a USB keyboard.
-
- **leds_pio -** This was used in lab 6 - not needed for Lab 7. This was used in lab 6.1 as a PIO output that was needed for the LED's to light up to show the sums of our accumulation.
-
- **timer_0 -** Used to send a signal so that NIOS II can check time passed. Used to send a signal so that NIOS II can check time passed.
-
- **spi_0** - SPI_0 is a module that provides a means for devices to communicate with each other using the Serial Peripheral Interface (SPI) protocol. The SPI interface facilitates the transfer of data between a master device and one or more slave devices using two communication channels, MOSI (Master Output Slave Input) and MISO (Master Input Slave Output). Depending on the specific mode of operation, the transfer of data can occur simultaneously in both directions. The SPI interface also includes a slave select line that enables the master device to select the target slave device for communication.
- **VGA_text_mode_controller -** the VGA_text_mode_controller is a PIO block that was created to provide the input and output signals that were to be used to display our data on the VGA. This PIO block contains two main important parts: the input/output signals and the Avalon-MM slave port which contains more specific input and output signals.
    - input/output signals: clock input (CLK), reset (RESET), Red[3:0], Blue[3:0]
      The VGA_text_mode_controller contains the inputs and outputs shown above. There is a clock signal that runs at 50 MHz (same as the one from the FPGA) and the reset signal. There are red, blue, and green signals that we used to help create our color palettes as well as to provide the correct colors to the associated registers and to color the foreground and backgrounds properly.
    - Avalon MM Slave Port Interface Signals: read, write, readdata, writedata, address, byteenable, chipselect
        - read (input) - this depicts when we want to perform the read operation
        - write (input) - this depicts when we want to perform the write operation
        - readdata (output) - data we want to read
        - writedata (input) - data we want to write
        - address (input) - address used for the read or write operation
        - byteenable (input) - identifies which byte(s) are being written

- chipselect (input) - identifies when we want to do a read or write operation

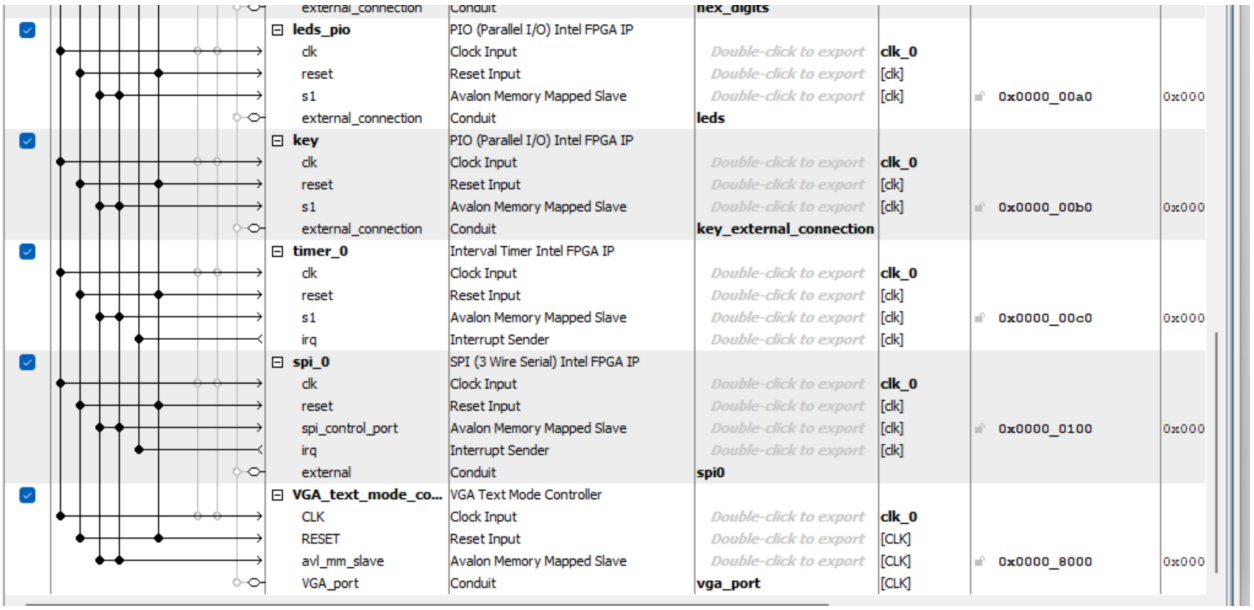


clk_0	Clock Source					
clk_in	Clock Input	clk	exported			
clk_in_reset	Reset Input	reset				
clk	Clock Output	Double-click to export	clk_0			
clk_reset	Reset Output	Double-click to export				
nios2_gen2_0	Nios II Processor					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0		
debug_reset_request	Reset Output	Double-click to export	[clk]			
debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_1000	0x000	
custom_instruction_m...	Custom Instruction Master	Double-click to export				
sdram	SDRAM Controller Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_0000	0x0bf	
wire	Conduit	sdram_wire				
sdram_pll	ALTPLL Intel FPGA IP					
inclk_interface	Clock Input	Double-click to export	clk_0			
inclk_interface_reset	Reset Input	Double-click to export	[inclk_interf...			
pll_slave	Avalon Memory Mapped Slave	Double-click to export	[inclk_interf...	0x0000_0060	0x000	
c0	Clock Output	Double-click to export	sdram_pll_c0			
c1	Clock Output	clk_sdram	sdram_pll_c1			
sysid_qsys_0	System ID Peripheral Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0078	0x000	

Lab 7.2 PIO blocks image 1

jtag_uart_0	JTAG UART Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0008	0x000	
irq	Interrupt Sender	Double-click to export	[clk]			
keycode	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0030	0x000	
external_connection	Conduit	keycode				
usb_irq	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0020	0x000	
external_connection	Conduit	usb_irq				
usb_gpx	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0040	0x000	
external_connection	Conduit	usb_gpx				
usb_rst	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0050	0x000	
external_connection	Conduit	usb_rst				
hex_digits_pio	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0090	0x000	
external_connection	Conduit	hex_digits				

Lab 7.2 PIO blocks image 2



Lab 7.2 PIO blocks image 3

****PIO blocks for Lab 7.2 are the same as the PIO block for Lab 7.2****

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. Which design is more efficient, what are the tradeoffs?

7.1:

LUT	**35698**
DSP	**0**
Memory (BRAM)	**50688**
Flip - Flop	**21696**
Frequency	**67.2 MHz**
Static Power	**97.22 mW**
Dynamic Power	**294.19 mW**
Total Power	**368.70 mW**

Design resources and statistics table - data from the compilation summary in Quartus

7.2:

LUT	**4617**
DSP	**0**
Memory (BRAM)	**193,536**
Flip - Flop	**2449**
Frequency	**75.22 MHz**
Static Power	**96.52 mW**
Dynamic Power	**62.93 mW**
Total Power	**181.9 mW**

Design resources and statistics table - data from the compilation summary in Quartus

6. Conclusion

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

Functionality:

During week 1 of the lab, we developed a monochrome graphics controller to link the NIOs II to the Avalon bus by utilizing 601 registers. Out of the 601 registers, 600 stored VRAM words, while the remaining register (601) functioned as a control register that allowed us to manage the color output using the VGA interface. We employed a fixed function for implementing the monochrome graphics controller as it did not require on-chip memory. Our screen comprised of a 30 x 80 matrix, with each block containing a glyph made up of 8 x 16 pixels. Accessing one of the 600 registers through the Avalon bus enabled us to draw the pixels and print four different characters on the screen.

For week 2 of the lab, we made modifications to the graphics controller developed in week 1 to enable color printing using a color palette register. Furthermore, we moved the 600 registers that held the characters to the on-chip memory. In lieu of a control register for managing colors, we utilized eight registers (0x800 - 0x807) to store a color palette, which acted as a look-up-table for each color. The lab included a total of 16 colors, with each of the eight palette registers containing 32-bit words. We differentiated the access to VRAM and the color palette by using a single address bit (e.g., ADDR[11]).

Problems we encountered and what we did to fix them:

During lab 7.1, we encountered an issue with our read and write operations for the 601 VRAM registers. Our memory did not have enough space, and too many memory blocks were in use. This was due to insufficient coverage of edge cases in our conditional statements for reading and writing, leading to the creation of shadow registers. Unfortunately, these shadow registers took up a significant amount of memory and caused an error during compilation. To overcome this problem, we enhanced our conditional statements to make them more comprehensive and robust.

Another issue we had occurred when our code was functioning correctly. The VGA cables was damaged and as a result, the wrong colors were being displayed so we switched VGA cables.

We also encountered indexing issues and on chip memory instantiation issues when transitioning from lab 7.1 to lab 7.2. We resolved this by unit testing our 7.1 lab first with on chip memory before moving our lab on to 7.2. We were also able to fix our indexing issues by tuning our math, formulas, and changing our registers to only hold 2 colors and 2 codes with inverse bits.

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

Incorporating 6.2, 7.1, and 7.2 into our Final Project would prove advantageous as we intend to utilize VRAM to store the display of our Legend of Zelda game. Additionally, we will employ the software and hardware from lab 6.2 to facilitate the movement of our sprite. Building upon

the key codes provided to us and the established physics of the ball.sv, we aim to further develop our game.

c. 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