ECE 385

Spring 22

Experiment #7

VGA Text Mode Controller with Avalon-MM Interface

Siddarth Iyer, Geitanksha Tandon

TA: Ruihao Yao (RY)

1. Introduction:

---

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

In this experiment, we design a simple text mode graphics controller interfaced with the Avalon Memory-Mapped bus to display one of 128 glyphs. It supports 80 columns and 30 rows with the capability of displaying a total of 2400 characters. Data regarding the characters to be printed is stored in the VRAM (Registers in lab 1, On-chip memory in lab 2) and accessed via the Avalon bus to write onto the VGA display. The graphic controller supported monochrome display in week 1 with the ability to set a singular foreground and background color, with capabilities extended in week 2 to display up to 16 colors via color palettes. The glyphs are stored as a bitmap with each glyph consisting of 8x16 pixels.

---

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

In lab 6.1 we had created a NIOS-II based system on the MAX10 device where the NIOS would be the system controller, and then the peripherals in the FPGA would handle higher-processor operations. In lab 6.2, we altered the system to integrate a USB host-controller interface (MAX2321E) that would communicate with our DE10-Lite board with the SPI protocol. In 6.2, we also used the DE10-Lite board's USB port to connect it to our VGA Monitor.

In lab 7, we had to support a text-mode graphical controller IP core on our platform designer. This needed to be connected to our Avalon bus (memory mapped) to support the 80x30 character mode on our VGA Monitor. We just built the IP and understood how to construct it. We needed to determine which glyphs needed to be drawn and how to draw it.

---

2. Written Description of Lab 7 System

a. Week 1 (Monochrome Text Display)

i. Written Description of the entire Lab 7 system

This is answered in the entire lab report - In the first week, we had to create a monochrome text display to track the glyphs via Font_rom and then print out sentences with these glyphs in monochrome colors. We created an IP module that would handle the text-mode graphics controller and support an 80 character mode.

---

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

The VGA Text Mode controller is connected to the Avalon memory-mapped bus and supports 80 columns and 30 rows. This corresponds to a total of 2400 characters. The data for each of these 2400 cells was stored in 600 32-bit registers with each register storing data for 4 cells. The registers are read from and written to via the Avalon bus. Each cell had a corresponding 8 bits of data, with 7 bits of 'code' corresponding to the 'Glyph code from IBM Codepage 437', and a singular invert bit. The 601st register stored 4 bit RGB values for the foreground and background color. Based on the current DrawX and DrawY from the VGA Controller module, the IP calculates which cell is currently being written, gets the 7 bit Glyph code, accesses the fontdata and determines which row and column of font data is currently being printed. Based on the fontdata and with the use of invert logic, the RGB value for the pixel is set as either the foreground or background.

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	**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

---

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

The VGA registers are read and written via the Avalon bus. The Avalon bus is implemented using platform designer. The avl interface module consisted of various signals: AVL_READ, AVL_WRITE, AVL_CS, AVL_BYTE_EN, AVL_ADDR, AVL_WRITEDATA, AVL_READDATA. It also consists of a 50 MHz clock. Therefore, the logic for the reading and writing is placed within an always_FF clock and triggered on the positive edge of the clock. The signal AVL_CS is a 1 bit chip select signal used to indicate that the memory is selected. Therefore the registers can only be read from or written to if AVL_CS is high. The AVL_READ and AVL_WRITE are used to indicate whether data is being read from or written to the registers, AVL_ADDR is a 10-bit address used to reference one of 601 VGA registers, and AVL_READDATA and AVL_WRITEDATA are 32 bit values used to store the data read from or to be written to the registers. If AVL_READ is high, the data from the VGA register is read based on the address specified by AVL_ADDR and stored in AVL_READDATA. AVL_BYTE_EN is a 4 bit value specifying which bytes of the 32-bit word are to be updated. If AVL_WRITE is high, based on the BYTE_EN data in AVL_WRITEDATA is written into the VGA register specified by AVL_ADDR.

```
always_ff @(posedge CLK) begin
        if (AVL_CS) begin
            if (AVL_READ)
                begin
                    AVL_READDATA <= LOCAL_REG[AVL_ADDR];
                end
            else if(AVL_WRITE)
                begin
                    case (AVL_BYTE_EN)

                        4'b1111: LOCAL_REG[AVL_ADDR]<=AVL_WRITEDATA;
                        4'b1100: LOCAL_REG[AVL_ADDR][31:16]<=AVL_WRITEDATA[31:16];
                        4'b0011: LOCAL_REG[AVL_ADDR][15:0]<=AVL_WRITEDATA[15:0];
                        4'b1000: LOCAL_REG[AVL_ADDR][31:24]<=AVL_WRITEDATA[31:24];
                        4'b0100: LOCAL_REG[AVL_ADDR][23:16]<=AVL_WRITEDATA[23:16];
                        4'b0010: LOCAL_REG[AVL_ADDR][15:8]<=AVL_WRITEDATA[15:8];
                        4'b0001: LOCAL_REG[AVL_ADDR][7:0]<=AVL_WRITEDATA[7:0];

                    endcase
                end


        end
end
```

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.

Name	Direction	Width	Description
read	Input	1	High when a read operation is to be performed.
write	Input	1	High when a write operation is to be performed.
readdata	Output	32	32-bit data to be read.
writedata	Input	32	32-bit data to be written.
address	Input	10	Address of the read or write operation.
byteenable	Input	4	4-bit active high signal to identify which byte(s) are being written.
chipselect	Input	1	High during a read or write operation.

---

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

The IP must determine the correct data to be printed based on the current DrawX and DrawY of the VGA. First, the cell from the 80*30 grid currently being operated on must be determined. DrawX and DrawY which corresponds to the coordinates in the 640*480 grid are first divided by 8 and 16 to obtain the coordinates in the 80*30 grid. This is done by accessing bits DrawX[9:3] and DrawY[9:4]. Since the data is stored sequentially in registers it is indexed in a row major order. To calculate the byte address we first multiply the column coordinate by 80 and add the row coordinate to this (DrawX[9:3]+DrawY[9:4]*80). Since each register corresponds to 4 bytes, we divide the byte address by 4 to get the final VRAM Address. Therefore the VRAM address corresponds to (DrawX[9:3]+DrawY[9:4]*80)[11:2]. To determine which of the 4 bytes we are dealing with within the 32-bit register, we use the Byte address modded with 4 or (DrawX[9:3]+DrawY[9:4]*80)[1:0]. Hence, using the calculated address, we receive 32 bits of

data from the register, and then use the 2 bit number to finally get the 8 bits of data for that specific cell. From the 8-bit number we use the bottom 7 bits which is the glyph code. The font rom consists of 128 sequentially stored characters, with each character taking up 16 rows. Within the 16 rows, to determine which row we are currently operating on we make use of DrawY%16 or the bottom 4 bits of DrawY. Therefore, to access the correct row within Font_ROM we multiply the 7-bit code with 16 and then add the bottom 4 bits of DrawY. This returns an 8-bit value. To determine which of the 8 bits we are currently working with we use DrawX%8 or the bottom 3 bits of DrawX to finally receive a singular bit of 1 or 0.

```
always_comb begin
        ac = DrawX[9:3] + (80*DrawY[9:4]);
        c4 = LOCAL_REG[ac[11:2]]; //reads from register based on address specified.

        case (ac[1:0]) //determines which byte from the 32-bit word
            2'b00: charac = c4[7:0];
            2'b01: charac = c4[15:8];
            2'b10: charac = c4[23:16];
            2'b11: charac = c4[31:24];
        endcase

        font_addr = (charac[6:0]*16 + DrawY[3:0]);
        bitval = font_data[3'b111 - DrawX[2:0]];  //Pixel bit
        invert = charac[7]; // invert bit

end
```

---

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

The control register stored 4-bit RGB values corresponding to the foreground and background colors. The invert bit was used to invert the color of the pixel. As per normal, if the pixel bit was a 0, the pixel color would be set to the background, and if it was 1, the pixel color would be set to the foreground. However, if the invert bit is a 1, it sets the color to foreground when the pixel bit is a 0, and background when the pixel bit is a 1. We take care of this logic by XORing the pixel bit with the invert bit. If the XOR yields a 1(invert = 0, pixel = 1 or invert = 1, pixel = 0), the RGB value is set to that of the foreground, and set to background otherwise.The foreground colors are set by accessing bits 24:21(red), 20:17(green), 16:13(blue) of register 600(Control register). Similarly the background color is determined by bits 12:9(red), 8:5(green), 4:1(blue)

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

```
always_ff @(posedge pixel_clk) begin

    if(!blank) begin
        red = 4'h0;
        green = 4'h0;
        blue = 4'h0;
    end

    else if (bitval^invert) begin
        red = LOCAL_REG[600][24:21];
        green = LOCAL_REG[600][20:17];
        blue = LOCAL_REG[600][16:13];
    end
    else begin
        red = LOCAL_REG[600][12:9];
        green = LOCAL_REG[600][8:5];
        blue = LOCAL_REG[600][4:1];
    end


end
```

---

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?

In week 1, data for each cell was stored within 600 registers. Each register stored data for 4 cells. However, data per cell for week 2, was expanded to 16 bits. This meant we wouldve required 1200 32-bit registers, which wouldve exhausted the FPGA logic units. For this reason, we use on-chip memory, with a true dual port implementation. Register implementation had multiple input and output ports making it easy to read and write, but onchip memory only had 4 read and write ports. Port a is used to manipulate AVL data while port B is used to access VRAM data. The addressability increases from 10 bits to 12 bits to account for the Pallete and the larger VRAM. Bit 11 of the address is used to determine whether the VRAM(if 0) or Pallete(if 1) is being accessed. The palette is implemented as 8 32-bit registers, and is read from and written to using logic similar to that used to read and write VGA registers from week 1. A mux is created with select bit as AVL_ADDR[11]to set the AVL_READDATA as either palletedata or data from VRAM.

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

The only modification in the platform designer was to change the addresability(size of address port of IP) from 10 bits to 12 bits to accommodate the additional VRAM and palette. In week 1, the VRAM consisted of 600 32-bit registers and an additional control register. Therefore, 10 bits was enough to address these registers. For week 2, since each cell now had 16-bits of data, only 2 cells could be stored per 32-bits. Therefore 2400 cells would require 1200 words. This would require 11 bits of addresability. However to incororporate the palette implementation we make use of 12 bits. After the VRAM is defined, unused space is left in the middle and the palette data is stored in memory such that the 11th bit is 1. This makes implementing the palette

much easier. Additionally, there is a delay in writing to memory since OCM is synchronous, which was not the case with registers.

Word Address Range	Byte Address Range	Description
0x000 - 0x4AF	0x0000 0000 - 0x0000 12BF	VRAM – 2 bytes per character, 2 characters per word. 80 column x 30 row. Data format is in raster order (one line at a time).
0x4B0 - 0x7FF	0x0000 12C0 - 0x0000 1FFF	Unused but reserved by Platform Designer
0x800 - 0x807	0x0000 2000 - 0x0000 201F	Palette- 8 words of 2 colors each, for 16-color palette
0x808 – 0xFFF	0x0000 2020 – 0x0000 3FFF	Unused but reserved by Platform Designer

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

Similar to week 1, the data to be printed is determined based on the current DrawX and DrawY of the VGA. Just like week 1, the byte address is first calculated as (DrawX[9:3]+DrawY[9:4]*80). However, now each VRAM address corresponds to 2 cells instead of 4 and the byte address is divided by 2 instead. The new VRAM Address is therefore (DrawX[9:3]+DrawY[9:4]*80)[11:1]. This address is inputted to port b of memory. The corresponding 32-bits of data is read from the output of port b of memory. Modding the byte address with 2 determines whether the upper 16 bits or lower 16 bits are being dealt with. Therefore the lowest bit of the byte address (DrawX[9:3]+DrawY[9:4]*80)[0] is used to select between the upper 16 and lower 16 bits. Of these 16 bits, [14:8] represents the 7-bit glyph code, and bit 15 represents the invert bit. Accessing Font_ROM is done similar to week 1 using DrawX, DrawY and the 7-bit glyph code.

4. Additional modifications necessary to support multicolored text.

The first step was to read and write data from and to the 8 32-bit palette registers. This logic was similar to that used to read and write VGA registers from week 1. The lowest 3 bits of AVL_ADDR determined which of the 8 Palletes were being written to or read from. Additionally the palettes would only be read to or written from when AVL_ADDR[11] was high.

```
always_ff @(posedge CLK) begin
      if (AVL_CS) begin
         if(AVL_WRITE && AVL_ADDR[11])
            begin
               case (AVL_BYTE_EN)
                  4'b1111: Palette[AVL_ADDR[2:0]]<=AVL_WRITEDATA;
                  4'b1100: Palette[AVL_ADDR[2:0]][31:16]<=AVL_WRITEDATA[31:16];
                  4'b0011: Palette[AVL_ADDR[2:0]][15:0]<=AVL_WRITEDATA[15:0];
                  4'b1000: Palette[AVL_ADDR[2:0]][31:24]<=AVL_WRITEDATA[31:24];
                  4'b0100: Palette[AVL_ADDR[2:0]][23:16]<=AVL_WRITEDATA[23:16];
                  4'b0010: Palette[AVL_ADDR[2:0]][15:8]<=AVL_WRITEDATA[15:8];
                  4'b0001: Palette[AVL_ADDR[2:0]][7:0]<=AVL_WRITEDATA[7:0];
               endcase
            end
         else if (AVL_READ && AVL_ADDR[11])
            begin
               palettedata <= Palette[AVL_ADDR[2:0]];
            end
      end
end
```

The next step was to set the pixel RGB value, based on the color index, and whether the color is to be set to foreground or background. Each index corresponded to a 4-bit value. The top three bits determined which of the 8 palettes were being used. Since each palette contained two colors, the LSB of the index is used to determine the color. For the foreground, the palette index was determined by bits 7:5 of the 16 bits, with bit 4 determining which of the 2 colors within the palette is to be set. Similarly, for the background, the palette index is determined by bits 3:1, with bit 0 determining which of the 2 colors within the palette.

```
always_ff @(posedge pixel_clk) begin
   if(!blank) begin
      red = 4'h0;
      green = 4'h0;
      blue = 4'h0;
   end
   else if (bitval^invert) begin
      if(charac[4]) begin
         red = Palette[charac[7:5]][24:21];
         green = Palette[charac[7:5]][20:17];
         blue = Palette[charac[7:5]][16:13];
         end
      else begin
         red = Palette[charac[7:5]][12:9];
         green = Palette[charac[7:5]][8:5];
         blue = Palette[charac[7:5]][4:1];
         end
   end
   else begin
      if(charac[0]) begin
         red = Palette[charac[3:1]][24:21];
         green = Palette[charac[3:1]][20:17];
         blue = Palette[charac[3:1]][16:13];
         end
      else begin
         red = Palette[charac[3:1]][12:9];
         green = Palette[charac[3:1]][8:5];
         blue = Palette[charac[3:1]][4:1];
         end
   end
end
```

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

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

5. Additional hardware/code to draw paletted colors

To draw palette colors, on the hardware side, we created 8 32-bit registers to store the palette data. These are written to, read from and implemented as explained above.

Additionally, we use c code to implement the palette functionality.

```c
void textVGAColorScreenSaver()
{
    char color_string[80];
    int fg, bg, x, y;
    textVGAColorClr();
    for (int i = 0; i < 16; i++)
    {
        setColorPalette (i, colors[i].red, colors[i].green, colors[i].blue);
    }
    while (1)
    {
        fg = rand() % 16;
        bg = rand() % 16;
        while (fg == bg)
        {
            fg = rand() % 16;
            bg = rand() % 16;
        }
        sprintf(color_string, "Drawing %s text with %s background", colors[fg].name, colors[bg].name);
        x = rand() % (80-strlen(color_string));
        y = rand() % 30;
        textVGADrawColorText (color_string, x, y, bg, fg);
        usleep (100000);
    }
}
```

This function is the main function to be called. It first sets the color palette based off a predetermined list, and then enters an infinite loop, selecting a randon position on the screen and setting the foreground and background to two different colors.

```
void setColorPalette (alt_u8 color, alt_u8 red, alt_u8 green, alt_u8 blue)
{
        int uppermask = color%2;
        alt_32 updatechar = vga_ctrl->palette[color/2]; //reads the 32 bit val
        if (uppermask) {
            updatechar &= 0xFE001FFF;
            updatechar |= (red << 21) | (green << 17) | (blue << 13);
        }
        else {
            updatechar &= 0xFFFFE001;
            updatechar |= (red << 9) | (green << 5) | (blue << 1);
        }
        vga_ctrl->palette[color/2] = updatechar;
}
```

This function is used to update the color palette by writing the 32 bit values in the corresponding memory location. Since there are 16 colors, the palette is determined and accessed by color/2. To determine, which of the 2 colors within the palette is being set, we use color%2. The 12 rgb bits are first set to 0 using bitmasking and then the new rgb values are written into memory by bit shifting and ORing.

```
void textVGADrawColorText(char* str, int x, int y, alt_u8 background, alt_u8 foreground)
{
    int i = 0;
    while (str[i]!=0)
    {
        vga_ctrl->VRAM[(y*COLUMNS + x + i) * 2] = foreground << 4 | background;
        vga_ctrl->VRAM[(y*COLUMNS + x + i) * 2 + 1] = str[i];
        i++;
    }
}
```

This function writes the character to be printed for each corresponding x and y coordinate

---

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.

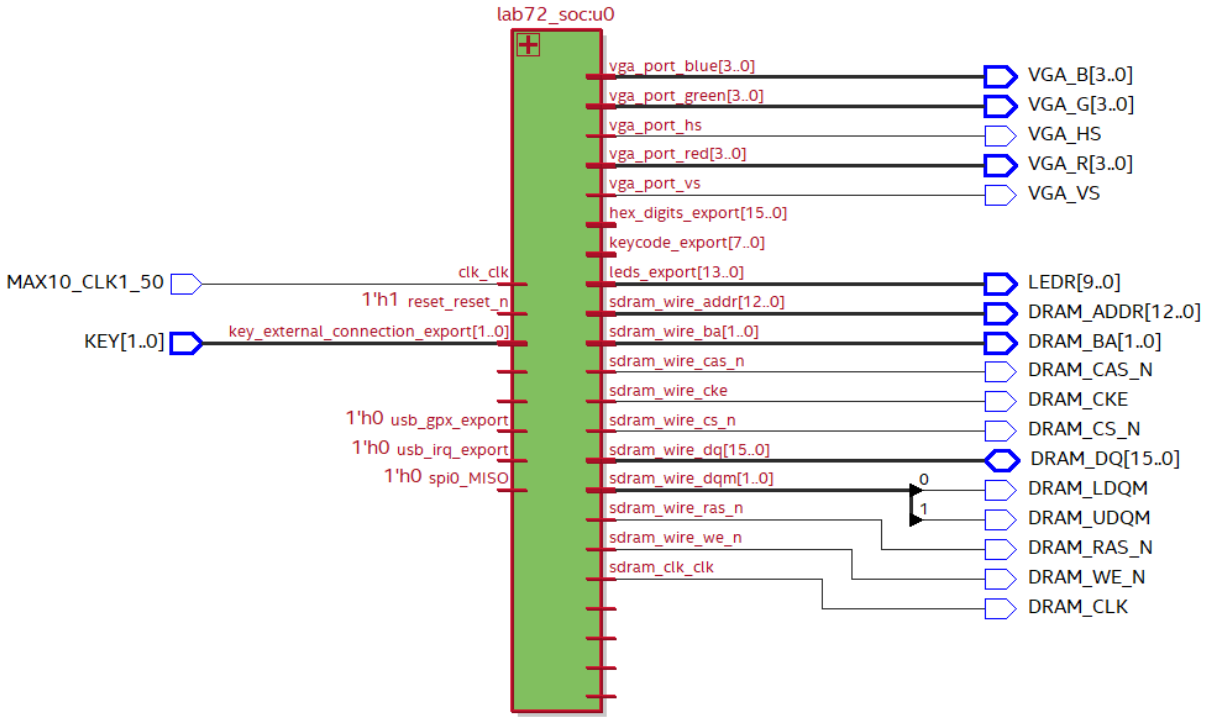
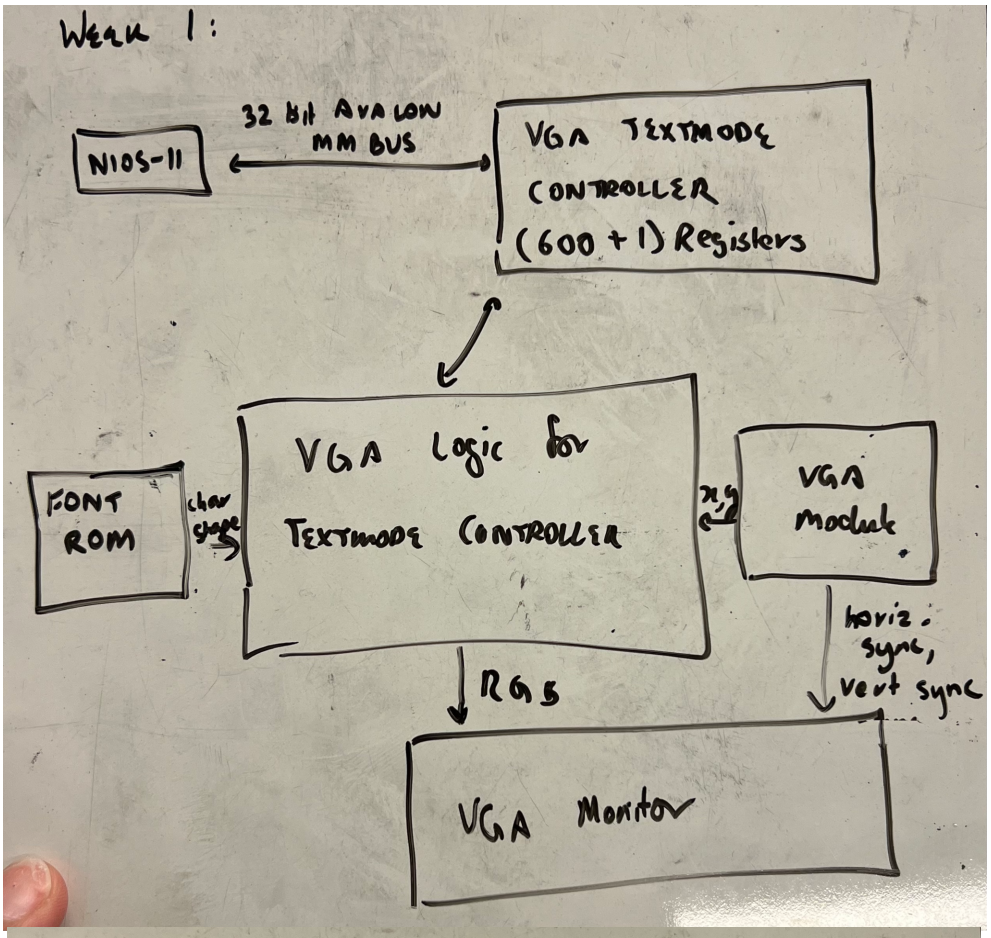
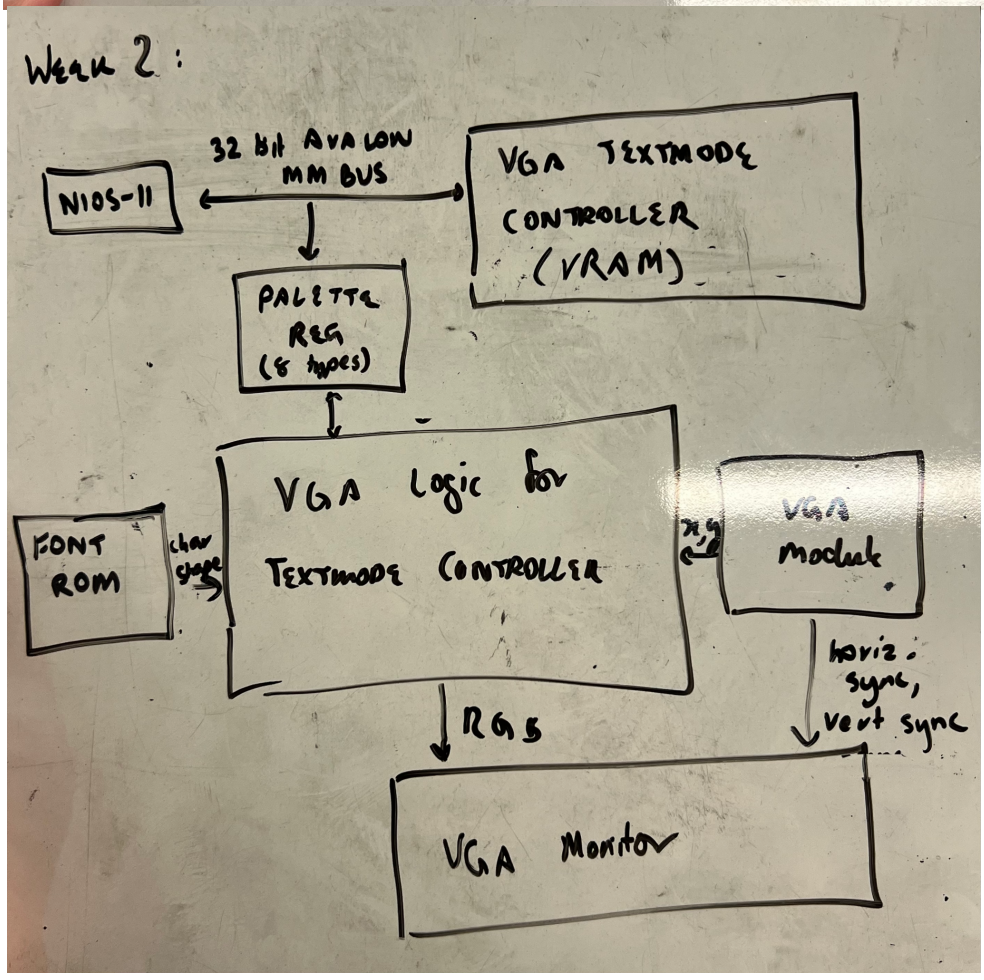


Figure: Diagram for the top level of both files: 7.1, 7.2.

The 2 weeks had the diagrams being very similar. The main difference was that the 601 registers in the week 1 translated into using VRAM and 8 palette registers instead. THe main changes in the output on the VGA Monitor is due to the functions that we wrote in C code.


Week 1:
NIOS-II
32 bit Avalon MM Bus
VGA Textmode Controller
(600 + 1) Registers
VGA Logic for Textmode Controller
Font ROM
char shape
x,y
VGA module
horiz. sync, vert sync
RGB
VGA Monitor


Week 2:
NIOS-II
32 bit Avalon MM Bus
VGA Textmode Controller
(VRAM)
Palette Reg
(8 types)
VGA Logic for Textmode Controller
Font ROM
char shape
x,y
VGA module
horiz. sync, vert sync
RGB
VGA Monitor

4. Module Descriptions

Module: vga_text_avl_interface.sv

Inputs: CLK,RESET,AVL_READ,AVL_WRITE,AVL_CS,[3:0] AVL_BYTE_EN,[11:0] AVL_ADDR,[31:0] AVL_WRITEDATA

Outputs: hs, vs, [31:0] AVL_READDATA,[3:0] red, green, blue

Description: This module instantiates the VRAM, vga_controller, font_rom, creates the palette, and handles the reading and writing of the palette. Using DrawX and DrawY it accesses the VRAM to determine the data to be printed, and accordingly sets the pixel color

Purpose: This module handles the read/write operation from memory to determine the color to be set on the display.

Module: VGA_controller.sv

Inputs: Clk, Reset

Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, DrawY

Description: This module generates the VGA signals by creating a 25MHz clock from the 50MHz clock given to using the always_ff block. It helps keep track of the coordinates of the pixel (X, Y coordinates) and describes the horizontal and vertical sync signals. It handles the blanking logic which helps set the value of the black color on the VGA.

Purpose: This module simulates the VGA to determine the current coordinate to be drawn and handles the vertical, horizontal sync and blank to draw pixels.

Module: font_rom.sv

Inputs: [10:0] addr

Outputs: [7:0] data

Description: This module defines the 128 glyphs stored in registers with each glyph represented as 16*8 bits.

Purpose: This module when given an address, outputs the 8 bits of data corresponding to a row in the font_data defined by that address

---

Module: ram.v

Inputs:[11:0] address_a, [11:0] address_b, [3:0] byteena_a, clock, [31:0] data_a, [31:0] data_b, wren_a, wren_b

Outputs: [31:0] q_a, [31:0] q_b

Description: This module is wizard-generated which creates a true dual port on chip memory element.

Purpose: This module is used to create the on-chip memory to store the VRAM data.

Module: Lab7.sv

Inputs: MAX10_CLK1_50, [1: 0] KEY, [9: 0] SW, [9: 0] LEDR, [7: 0] HEX0, HEX1,HEX2,HEX3,HEX4,HEX5, DRAM_CLK, DRAM_CKE, DRAM_ADDR,[1: 0] DRAM_BA, [15: 0] DRAM_DQ

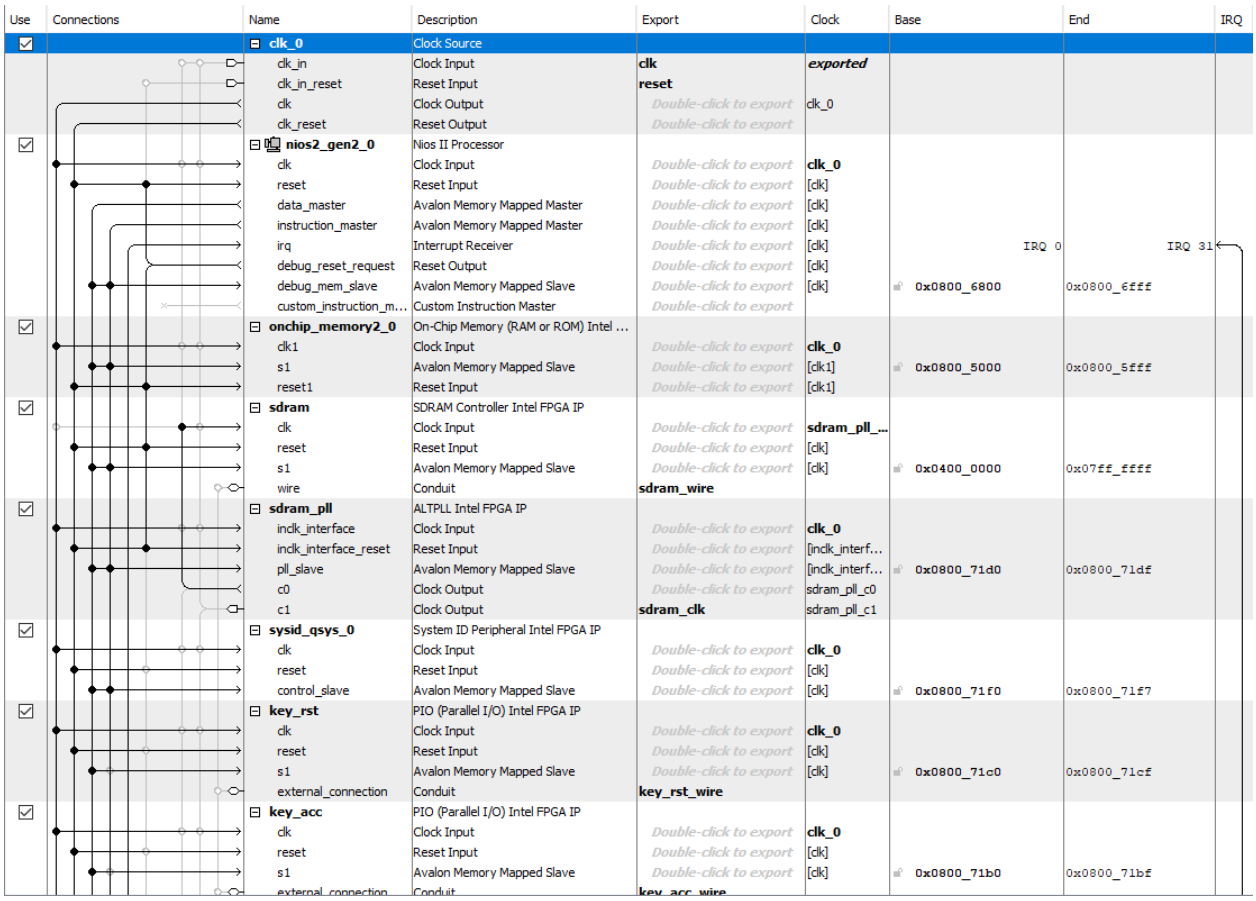
Outputs: DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS,VGA_VS, VGA_R, VGA_G, VGA_B

Inout: [15: 0]   ARDUINO_IO, ARDUINO_RESET_N

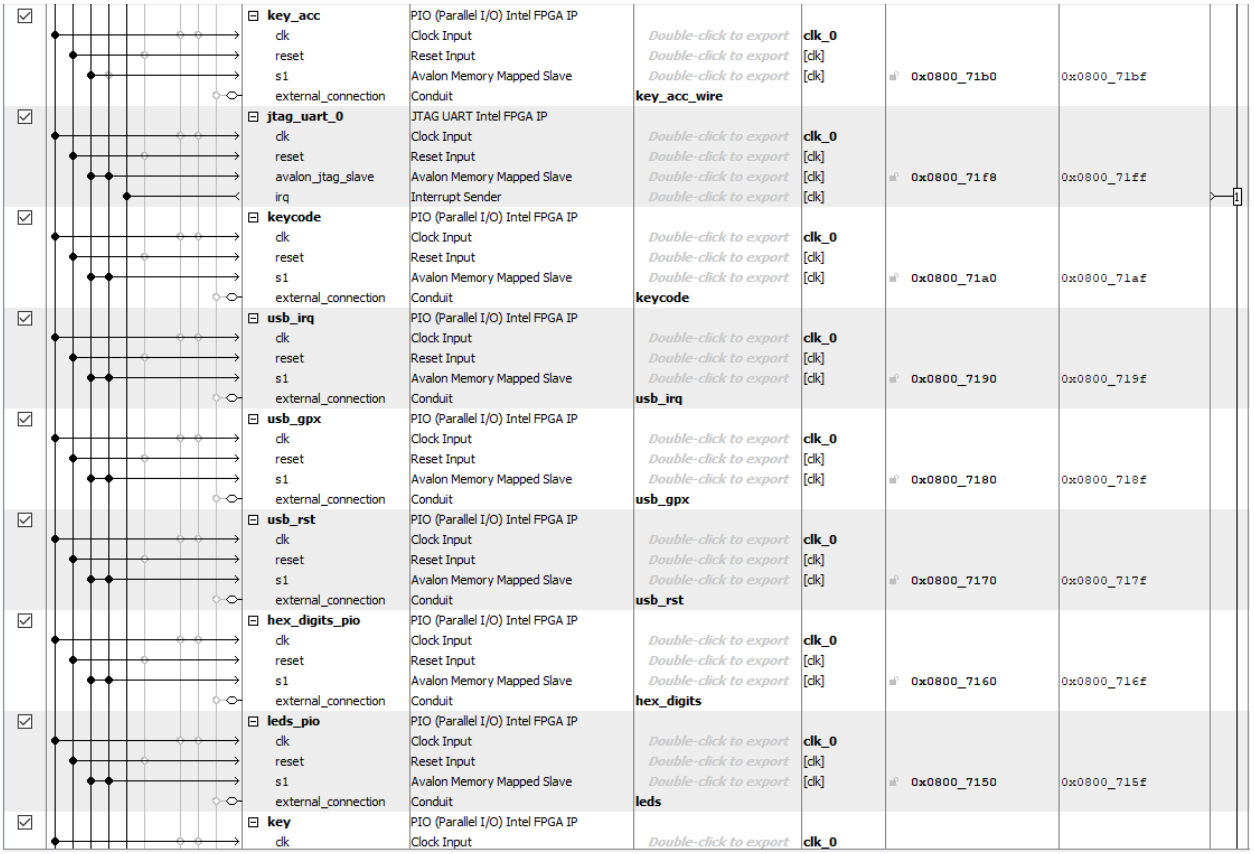
Description: This module is the top-level entity for lab 7, in which the according HEX and Switch connections are made and the lab7_soc is instantiated.

Module: lab72_soc.v

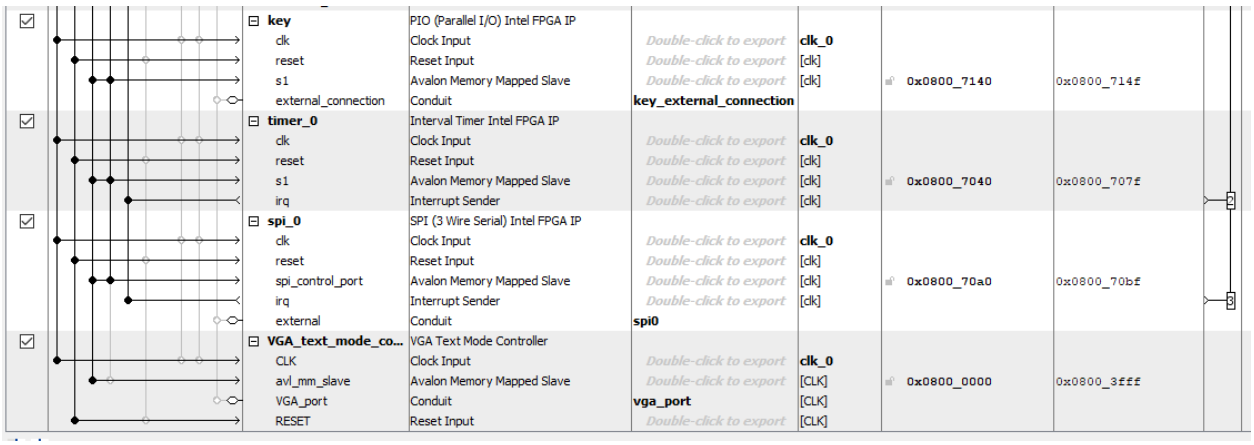
Only used components have been listed. Additional components are from lab 6.2



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☑		**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		IRQ 31
		debug_reset_request	Reset Output	*Double-click to export*	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_6800	0x0800_6fff	
		custom_instruction_m...	Custom Instruction Master	*Double-click to export*				
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel ...					
		clk1	Clock Input	*Double-click to export*	**clk_0**			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0800_5000	0x0800_5fff	
		reset1	Reset Input	*Double-click to export*	[clk1]			
☑		**sdram**	SDRAM Controller Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**sdram_pll_...**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0400_0000	0x07ff_ffff	
		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...	0x0800_71d0	0x0800_71df	
		c0	Clock Output	*Double-click to export*	sdram_pll_c0			
		c1	Clock Output	**sdram_clk**	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]	0x0800_71f0	0x0800_71f7	
☑		**key_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]	0x0800_71c0	0x0800_71cf	
		external_connection	Conduit	**key_rst_wire**				
☑		**key_acc**	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]	0x0800_71b0	0x0800_71bf	
		external_connection	Conduit	**key_acc_wire**				



key_acc	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]	0x0800_71b0	0x0800_71bf	
external_connection	Conduit	**key_acc_wire**				
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]	0x0800_71f8	0x0800_71ff	
irq	Interrupt Sender	Double-click to export	[clk]			1
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]	0x0800_71a0	0x0800_71af	
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]	0x0800_7190	0x0800_719f	
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]	0x0800_7180	0x0800_718f	
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]	0x0800_7170	0x0800_717f	
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]	0x0800_7160	0x0800_716f	
external_connection	Conduit	**hex_digits**				
leds_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]	0x0800_7150	0x0800_715f	
external_connection	Conduit	**leds**				
key	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	**clk_0**			



key	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]	0x0800_7140	0x0800_714f	
external_connection	Conduit	**key_external_connection**				
timer_0	Interval Timer 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_7040	0x0800_707f	
irq	Interrupt Sender	Double-click to export	[clk]			2
spi_0	SPI (3 Wire Serial) Intel FPGA IP					
clk	Clock Input	Double-click to export	**clk_0**			
reset	Reset Input	Double-click to export	[clk]			
spi_control_port	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800_70a0	0x0800_70bf	
irq	Interrupt Sender	Double-click to export	[clk]			3
external	Conduit	**spi0**				
VGA_text_mode_co...	VGA Text Mode Controller					
CLK	Clock Input	Double-click to export	**clk_0**			
avl_mm_slave	Avalon Memory Mapped Slave	Double-click to export	[CLK]	0x0800_0000	0x0800_3fff	
VGA_port	Conduit	**vga_port**	[CLK]			
RESET	Reset Input	Double-click to export	[CLK]			

Component	Name	Description
Nios-II Processor	nios2_gen2_0	This is our 32-bit CPU - the primary controller that is controlled by C, a high-level programming language. It is used to control the peripherals that primarily handle data and do not require fast processing times, and only need to transmit data.
SDRAM	sdram	The SDRAM (Synchronous Dynamic RAM) is used because of the limited availability of On-Chip memory.
Clock	clk_0	This module acts as the 50MHz clock from the FPGA which outputs the signal clk which can be used by other modules
SDRAM PLL	sdram_pll	The SDRAM is not on the FPGA and so needs a synchronous clock which is used in this module.
Timer	Timer_0	This is used for the NIOS II to keep track of elapsed time, by sending a signal with frequency 1000 Hz
SPI (W2)	spi_0	The SPI (System Peripheral Interface) lets us interact with USB Peripherals like the Keyboard, and mouse if necessary. It handles transfer of data between the NIOSII and the slave peripheral via MISO and MOSI.
JTAG UART (W2)	jtag_uart	This allows character movement between the Computer and the FPGA, enabling transferring of text which avoids the slowing down of the CPU for tasks that do not require high processing powers.
VGA Text Mode Controller IP	VGA_text_mode_controller_0	This IP is used to implement the Avalon Memory Mapped bus and control the VGA via hs, vs, RGB values

5. Design Resources and Statistics

Lab 7.1

LUT	35703
DSP	0
Memory (BRAM)	11264
Flip-Flop	21698
Frequency	67.21 MHz
Static Power	97.24 mW
Dynamic Power	249.24 mW
Total Power	368.74 mW

Lab 7.2

LUT	6034
DSP	0
Memory (BRAM)	49664
Flip-Flop	2870
Frequency	73.1 MHz
Static Power	96.6 mW
Dynamic Power	79.95 mW
Total Power	196.32 mW

In week 1 we implemented the VRAM using registers, while in week 2 we used OCM. This reflects in the statistics. Comparing the LUTs and flip-flops used, it is evident that week 1 required significantly more elements. This makes sense since there were 601 32-bit registers, and each bit is stored in a flip-flop. The BRAM for week 2 is significantly higher than week 1, since week 2 made use of on-chip memory and also required 1200 32-bit storage elements.

Since, week 2 requires significantly less LUTs and Flip-flops, the total power is also significantly lower. Based on statistics, it appears week 2 implementation is a lot more efficient. The only trade off is lack of input output ports from each of the registers, and additional wait states to write data into the synchronous onchip memory.

---

6. Conclusion

In conclusion, we have successfully created a simple text mode graphics controller interfaced with the Avalon Memory-Mapped bus to display glyphs. This lab has solidified our understanding with working with the NIOS II SoC. It gives us a starting place for future graphics rendering required for the final project. We have also implemented the Avalon Memory mapped Bus which is useful for data access and manipulation. This along with lab 6 gives us a detailed understanding to use peripherals such as keyboard, VGA monitor.
