ECE 385

Fall 2022

Experiment #7

VGA Text Mode Graphics with Avalon-MM Interface

Ravi Thakkar, Siraj Khogeer

TA: Gavin Wu

11-11-2022

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.

In this lab, we are aiming to create a VGA graphic controller that will allow us to print text with different colors and backgrounds onto a VGA monitor's display. The controller is connected to the Avalon Memory-Mapped bus. Our display is supposed to have 80 columns and 30 rows, and each cell contains one sprite, or a text character. For the VGA graphic controller, data from the VRAM is read to get the color and character from the font_rom file, and then output the character onto the screen. For the two weeks of this lab, the major change is how we are implementing the VRAM and how that affects our capabilities so as to allow us to have multiple colors on the screen for Lab 7.2 in comparison to having a monochrome display for Lab 7.1. For Lab 7.1, we used registers to store data as our implementation for the VRAM, but for Lab 7.2, we used on-chip memory as our implementation of the VRAM, which allowed us to palletize the colors so that we are able to write with different colors and backgrounds. This lab sets us up for more complex VGA drawing that can be used for the final project.

In Lab 6.2, we had a ColorMapper module that took care of coloring in the background and the ball that we were controlling through the keyboard by tracking the position of the center of the ball and filling in pixels accordingly. This approach strictly relied on hardware, but the approach that we take in Lab 7 builds on top of that and allows us to use software to control color assignment where the software is writing data into the registers or the on-chip memory, and subsequently the VGA controller takes that data and the display is updated accordingly.

Written Description of Lab 7 System

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

The SoC that we designed for Lab 7 was built on top of the Lab 6 SoC that we created on Platform Designer, using the 'Introduction to NIOS II and QSys' document. This means that our SoC has IP blocks including the NIOS-II CPU and peripherals such as the SDRAM, the JTAG UART, and the PIO blocks such as the USB and LED ones that we didn't utilize in this lab. The new addition for IP blocks in this lab was the 'VGA Text Mode Controller IP', which will be explained in the next section.

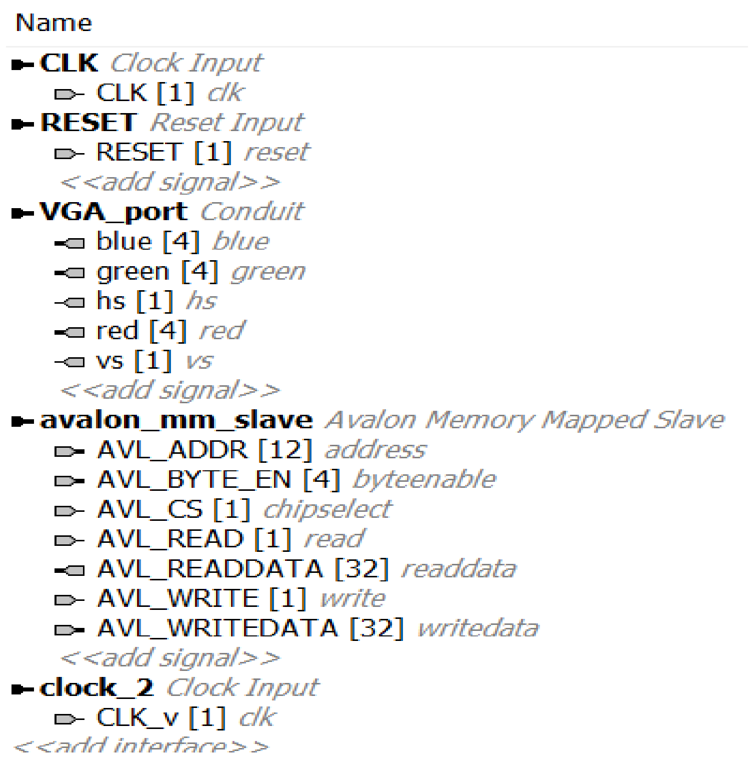
Since our implementation of the screen supports 80 columns and 30 rows, that allows us to have a total of 80*30 = 2,400 sprites, or characters, on the screen at the same time. That means for Week 1, where we were relying on registers as our implementation of the VRAM, we need 600 thirty-two bit registers, each that can hold data for four sprites, and we will need an extra control register, bringing the total number of registers to 601. Below shows how the data for four sprites is encoded within each register.

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

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

This is a custom IP core that we created to address the requirement of having an interface in between the CPU and VRAM. To address this, a set of signals through an Avalon Memory Mapped Slave are used. The IP takes care of the frame buffer, drawing, and the palettes. The IP stores the data for 4 characters in memory/registers, and then uses that with the palette to draw the screen. The IP block has the following signals:



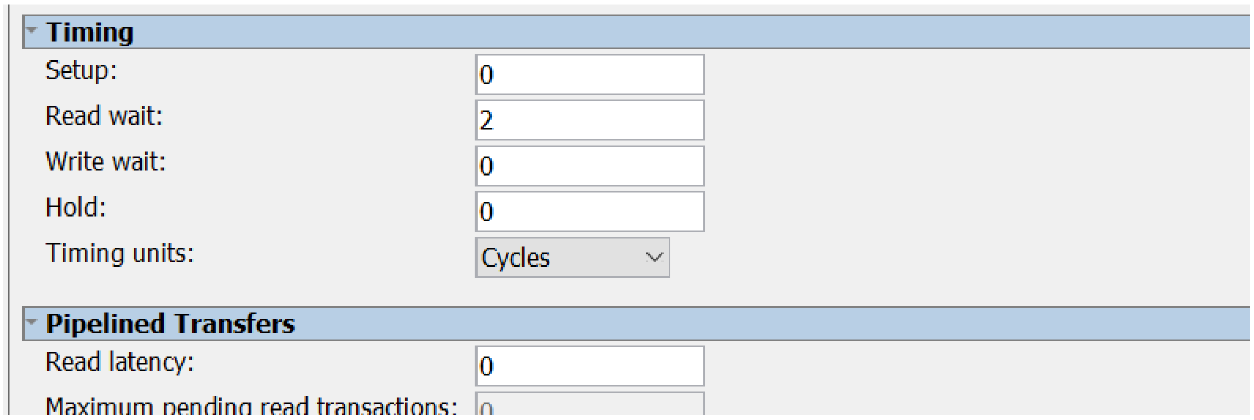
```
Name
CLK Clock Input
    CLK [1] clk
RESET Reset Input
    RESET [1] reset
    <<add signal>>
VGA_port Conduit
    blue [4] blue
    green [4] green
    hs [1] hs
    red [4] red
    vs [1] vs
    <<add signal>>
avalon_mm_slave Avalon Memory Mapped Slave
    AVL_ADDR [12] address
    AVL_BYTE_EN [4] byteenable
    AVL_CS [1] chipselect
    AVL_READ [1] read
    AVL_READDATA [32] readdata
    AVL_WRITE [1] write
    AVL_WRITEDATA [32] writedata
    <<add signal>>
clock_2 Clock Input
    CLK_v [1] clk
<<add interface>>
```

The Avalon Memory Mapped slave signals are based on the following table:

Table 1. Avalon-MM Slave Port Interface Signals

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.

We also delay the read signal to ensure that the signal is completely synchronized before we try to read it. We found that applying a wait of two clock cycles for the read signal timing worked best for our implementation of the lab.



Timing

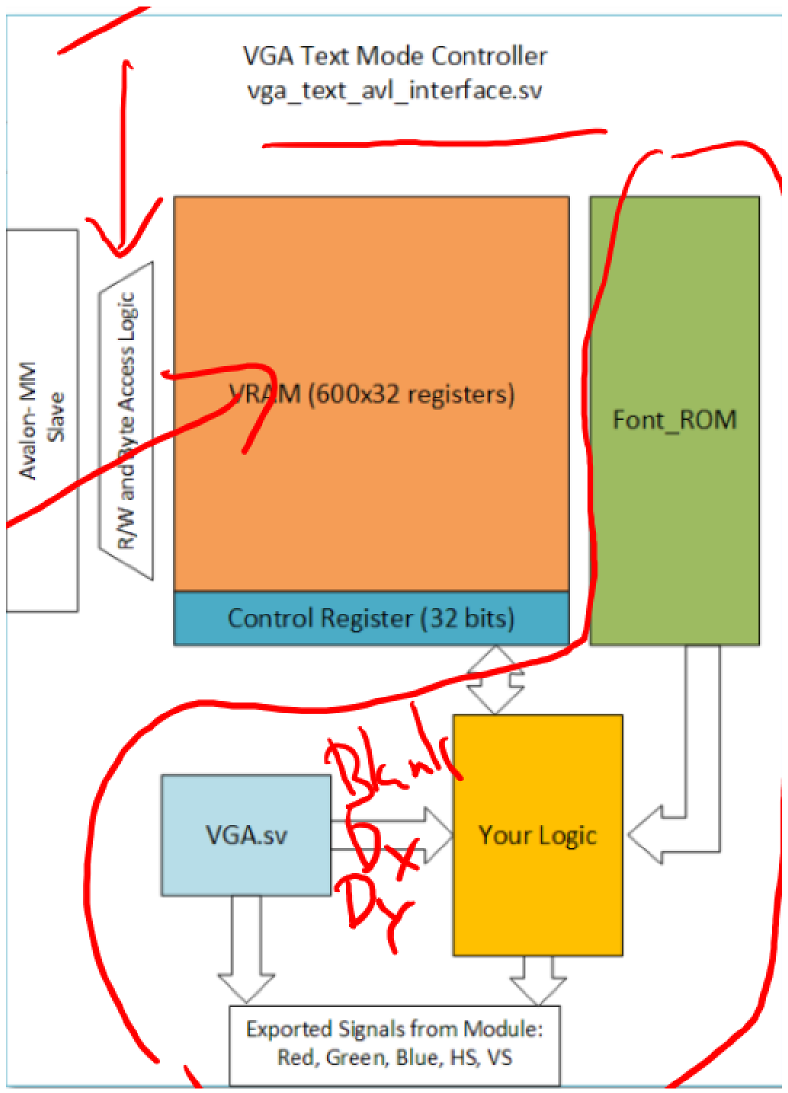
Setup:	0
Read wait:	2
Write wait:	0
Hold:	0
Timing units:	Cycles

Pipelined Transfers

Read latency:	0
Maximum pending read transactions:	0

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

We have 600 thirty-two-bit registers that each hold data for four sprites each, and 1 register used for control. The following graphic illustrates how each component of the module vga_text_avl_interface interacts with each other. Essentially, the AVL_ADDR points to one of the 601 registers and then the Avalon Bus writes the data into that register. Code shown below


VGA Text Mode Controller
vga_text_avl_interface.sv
Avalon- MM Slave
R/W and Byte Access Logic
VRAM (600x32 registers)
Control Register (32 bits)
Font_ROM
VGA.sv
Your Logic
Blank
Dx
Dy
Exported Signals from Module: Red, Green, Blue, HS, VS

```
if (AVL_CS & AVL_WRITE) begin
                LOCAL_REG[7:0]   = AVL_BYTE_EN[0] ? AVL_WRITEDATA[7:0]   : LOCAL_REG[AVL_ADDR][7:0];
                LOCAL_REG[15:8]  = AVL_BYTE_EN[1] ? AVL_WRITEDATA[15:8]  : LOCAL_REG[AVL_ADDR][15:8];
                LOCAL_REG[23:16] = AVL_BYTE_EN[2] ? AVL_WRITEDATA[23:16] : LOCAL_REG[AVL_ADDR][23:16];
                LOCAL_REG[31:24] = AVL_BYTE_EN[3] ? AVL_WRITEDATA[31:24] : LOCAL_REG[AVL_ADDR][31:24];
        end else if (AVL_CS & AVL_READ) begin
                                AVL_READDATA = LOCAL_REG[AVL_ADDR][31:0];
end
```

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 drawing process is split into two parts. Getting the current character index, and then the color. Each register has 4 characters, so the formula "LOCAL_REG[(DrawX/32)+(DrawY/16)*20][X:X]" is used to get the specific register that hold the current 4-character chunk that is being drawn. The DrawX/32 find which horizontal register it is, and the DrawY/16*20 increments the count by 20 after each character is drawn.

DrawX%32/8 is then used to find which of the 4 characters is currently being drawn. Mod 32 is used to isolate the coordinates into 32 bits which corresponds to 1 register. This is then divided by 8 to get a value from 0 to 3 which corresponds to which character is being drawn. Finally, the address of the font rom is decided using the formula addr = 16 * LOCAL_REG[(DrawX/32)+(DrawY/16)*20][DrawX%32/8*8+6:DrawX%32/8*8] + DrawY%16. This comes from the architecture of the font_rom. The index of the character is multiplied by 16 to get the corresponding font as each character has 16 lines. DrawY%16 is then added because the font_rom holds the data for each row of the character below each other, so this is used to draw the correct row of the character pixels. This font_rom then gives us data which holds the pixel data of the character being drawn. An if statement data[8-DrawX%32%8] == 1 is then used to determine if the current pixel drawn should be foreground or background colored. DrawX%32%8 is used to determine the horizontal coordinate of the pixel within the character. 8-DrawX%8 is used because that is the way the font_rom is implemented, which has to do with little and big endian.

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

The inverse color bit is implemented utilizing an if statement: LOCAL_REG[DrawX/32+(DrawY/16)*20][((DrawX%32)/8*8+7)]==0. This looks at the current character being drawn, and if the inverse bit is 0, the foreground color and background are kept constant, but if it is 1, then during drawing, the color for the foreground and background are flipped. This is essentially a case statement before the drawing process.

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?

Our design used a 2-port RAM on chip memory. This memory was 32-bits per word and held 1200 words, where each word held the data for 2 characters, their color palette indexes, and an inverse bit for both. To implement this we had one read/write port dedicated to the AVL bus, and another for the drawing circuit. The AVL address was connected to the OCM address as well as the byte enable and other variables. Note that the wren for the OCM depended on whether the AVL_ADDR[11] was 1 or 0. If it was 0 then the AVL_Write signal was passed through, otherwise the wren was set to low and the palette was written to instead. We also ran this OCM at 200MHZ by passing through another clock signal from the SDRAM_PLL, this was to reduce glitches. The other port was dedicated to the drawing circuit which was read only with the address as (DrawX/16)+(DrawY/16)*40 which gave the data for the current character being drawn, this was stored in a temp variable. All of this increased the required address size to 12 bits.

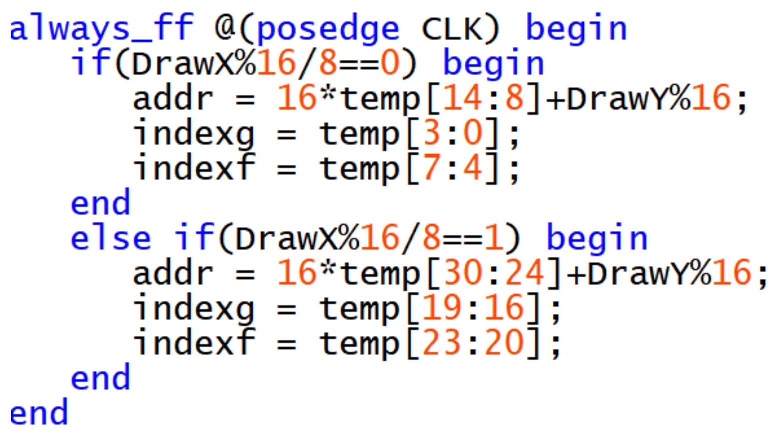
2. Corresponding modifications to the Platform Designer IP (e.g. Part Editor).

The main difference between the two was that we added an extra clock signal that ran at 200 mhz to run the OCM. This came from the sdram_pll. We also needed to make sure that the SOC included the increased AVL_ADDR from 10 bits to 12-bits.

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

To get the word data (DrawX/16)+(DrawY/16)*40 was used to get the data from the OCM into the variable temp. addr = 16*temp[DrawX%16/8*8+6:DrawX%16/8*8]+DrawY%16 is then used to get the address for the font_rom. Note that DrawX%16/8*8 is used to get the location of the character index in the current word data. Depending on which character is being drawn, the foreground and background indexes are then stored in indexg and indexf.

data[8-DrawX%16%8] == 1 is then used to determine whether the pixel should be drawn in conventional foreground or background. Note that an if statement looking at the inverse bit is also used to determine which color to draw the pixel with: temp[((DrawX%16)/8*16+15)]==0



```
always_ff @(posedge CLK) begin
   if(DrawX%16/8==0) begin
      addr = 16*temp[14:8]+DrawY%16;
      indexg = temp[3:0];
      indexf = temp[7:4];
   end
   else if(DrawX%16/8==1) begin
      addr = 16*temp[30:24]+DrawY%16;
      indexg = temp[19:16];
      indexf = temp[23:20];
   end
end
```

4. Additional modifications necessary to support multicolored text.

To support multicolored text a palette is used to draw each pixel. Each character data holds the foreground and background color indexes which are then used to determine the foreground and background colors. The drawing algorithm is then used to draw the pixel as shown below. Note that this does not include the inverse bit. Note that the palette structure does not match the provided palette due to bugs.

```
if(data[8-DrawX%16%8] == 1) begin
   red = colorf[11:8];
   green = colorf[7:4];
   blue = colorf[3:0];
end else begin
   red = colorg[11:8];
   green = colorg[7:4];
   blue = colorg[3:0];
end
```

5. Additional hardware/code to draw paletted colors

To be able to draw utilizing palletized colors two new things were needed. The first is the way to write to the palette. The palettes were stored on FPGA registers, and there were 16 total. To write the palette registers as a similar process was used for lab 7.1, this is shown below. If the MSB of the AVL_ADDR is 1 then the circuit would write to the palette specified by the 3 LSB bits of the address. This palette is then used in a combinational unique case statement to determine the foreground and background colors. This is shown below. In this case, the palette register holds the first color in its first two bytes, and the second color in its second two, making a 32-bit word. This is then uniquely cased to set the colors to be drawn.

```
 if(AVL_ADDR[11]&AVL_WRITE&AVL_CS) begin

    PIN[7:0]    = AVL_BYTE_EN[0] ? AVL_WRITEDATA[7:0]    : pallete[AVL_ADDR[2:0]][7:0];
    PIN[15:8]   = AVL_BYTE_EN[1] ? AVL_WRITEDATA[15:8]   : pallete[AVL_ADDR[2:0]][15:8];
    PIN[23:16]    = AVL_BYTE_EN[2] ? AVL_WRITEDATA[23:16]  : pallete[AVL_ADDR[2:0]][23:16];
    PIN[31:24]   = AVL_BYTE_EN[3] ? AVL_WRITEDATA[31:24]  : pallete[AVL_ADDR[2:0]][31:24];
    pallete[AVL_ADDR[2:0]] = PIN;

 end

always_comb begin
   unique case(indexf)
      0 :   colorf = pallete[0][11:0];
      1 :   colorf = pallete[0][27:16];
      2 :   colorf = pallete[1][11:0];
      3 :   colorf = pallete[1][27:16];
      4 :   colorf = pallete[2][11:0];
      5 :   colorf = pallete[2][27:16];
      6 :   colorf = pallete[3][11:0];
      7 :   colorf = pallete[3][27:16];
      8 :   colorf = pallete[4][11:0];
      9 :   colorf = pallete[4][27:16];
      10 :  colorf = pallete[5][11:0];
      11 :  colorf = pallete[5][27:16];
      12 :  colorf = pallete[6][11:0];
      13 :  colorf = pallete[6][27:16];
      14:   colorf = pallete[7][11:0];
      15 :  colorf = pallete[7][27:16];
   endcase
   unique case(indexg)
      0 :   colorg = pallete[0][11:0];
```

Block Diagram

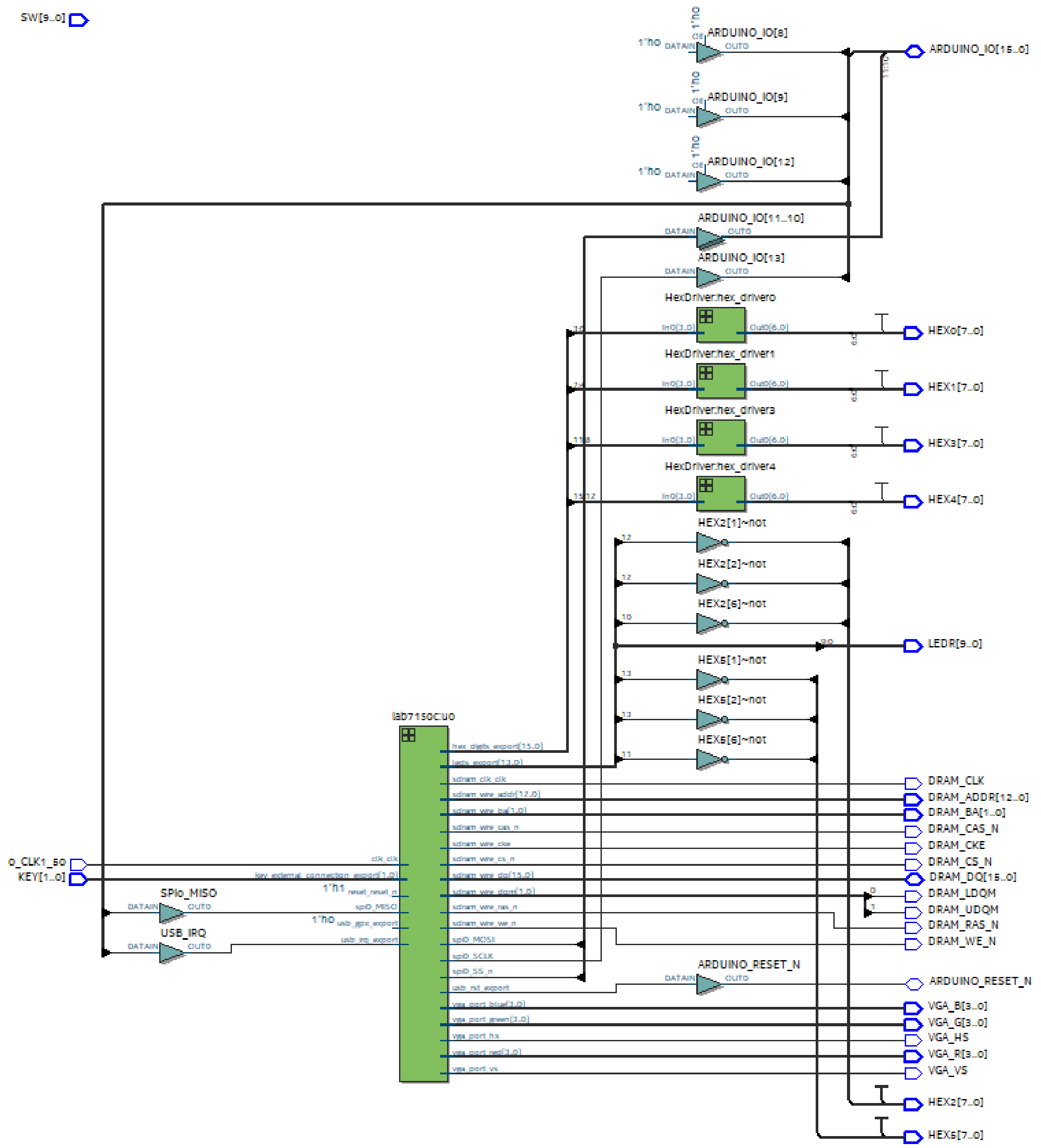
a. This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module.
b. Note that depending on your layout of the registers inside your main module, the Quartus view may be illegible, in which case you should 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.

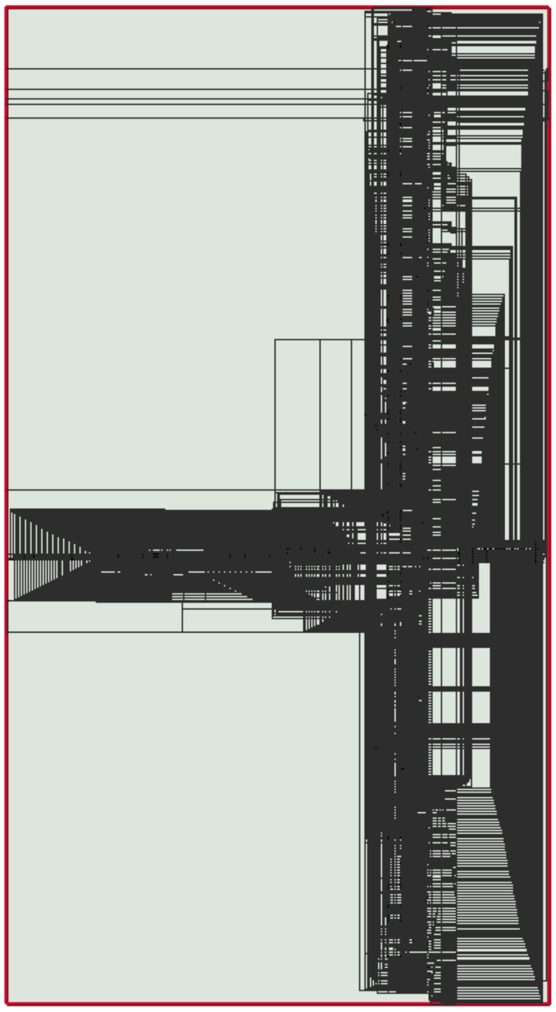
c. You should have block diagrams for both the Week 1 and Week 2 portions, a good setup is to show the common components (e.g. the SoC setup) first and then show diagrams for both the Week 1 and Week 2 VGA controller components.

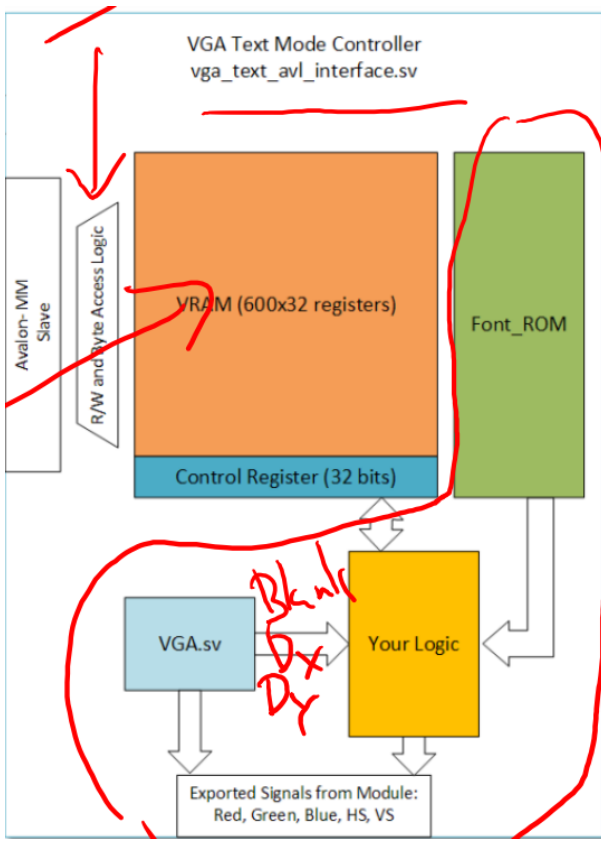
d. If your design has a state machine, you should include a State Diagram as well.

<u>Week 1 Top Level</u>



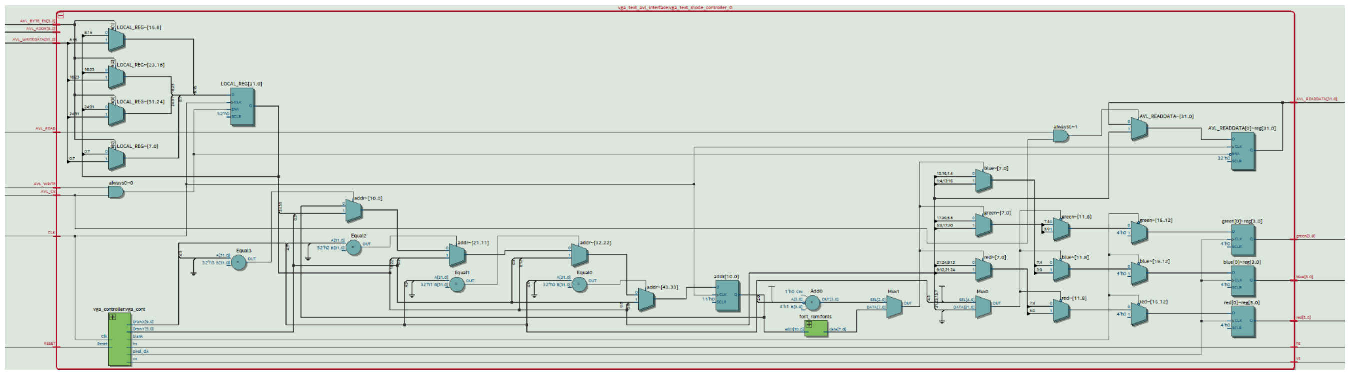
Week 1 vga_text_avl_interface.sv Block Diagram





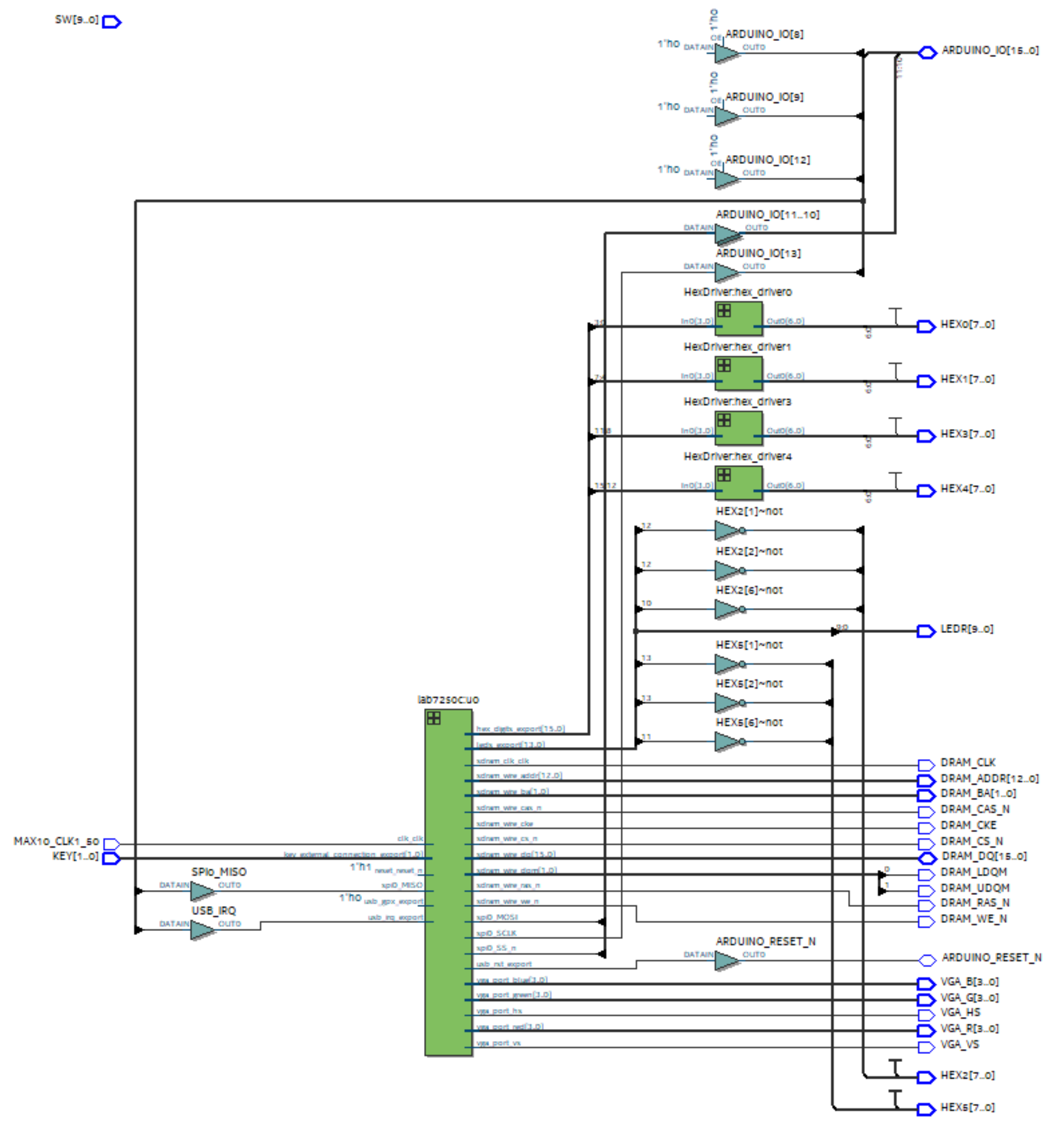
VGA Text Mode Controller
vga_text_avl_interface.sv
Avalon- MM Slave
R/W and Byte Access Logic
VRAM (600x32 registers)
Control Register (32 bits)
Font_ROM
VGA.sv
Your Logic
Blank
Dx
Dy
Exported Signals from Module: Red, Green, Blue, HS, VS

Since this graphic is very hard to understand, a simplified version of this diagram would be the diagram provided to us in the lectures.

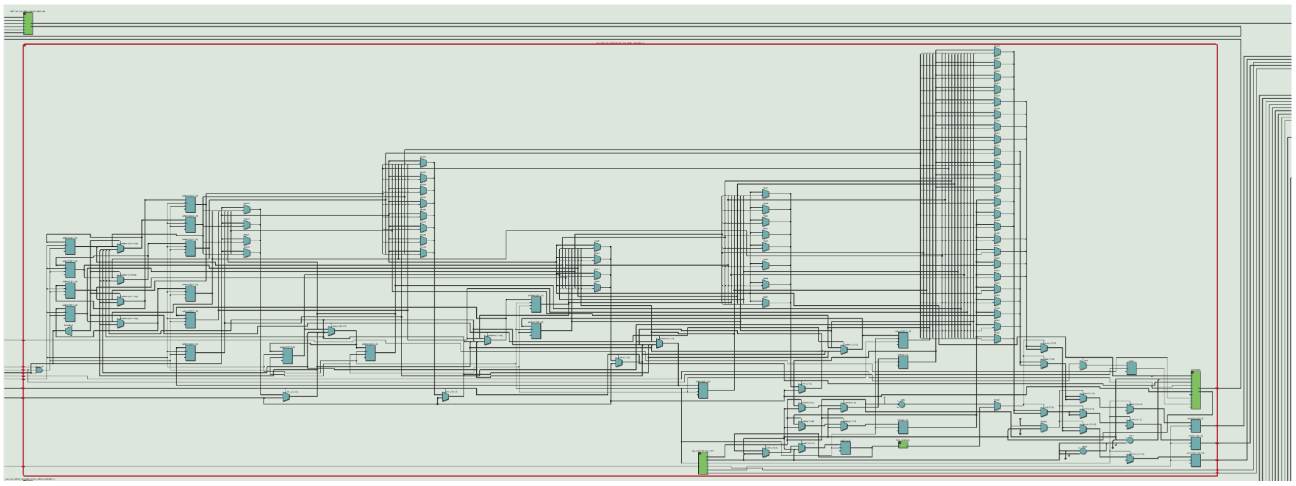


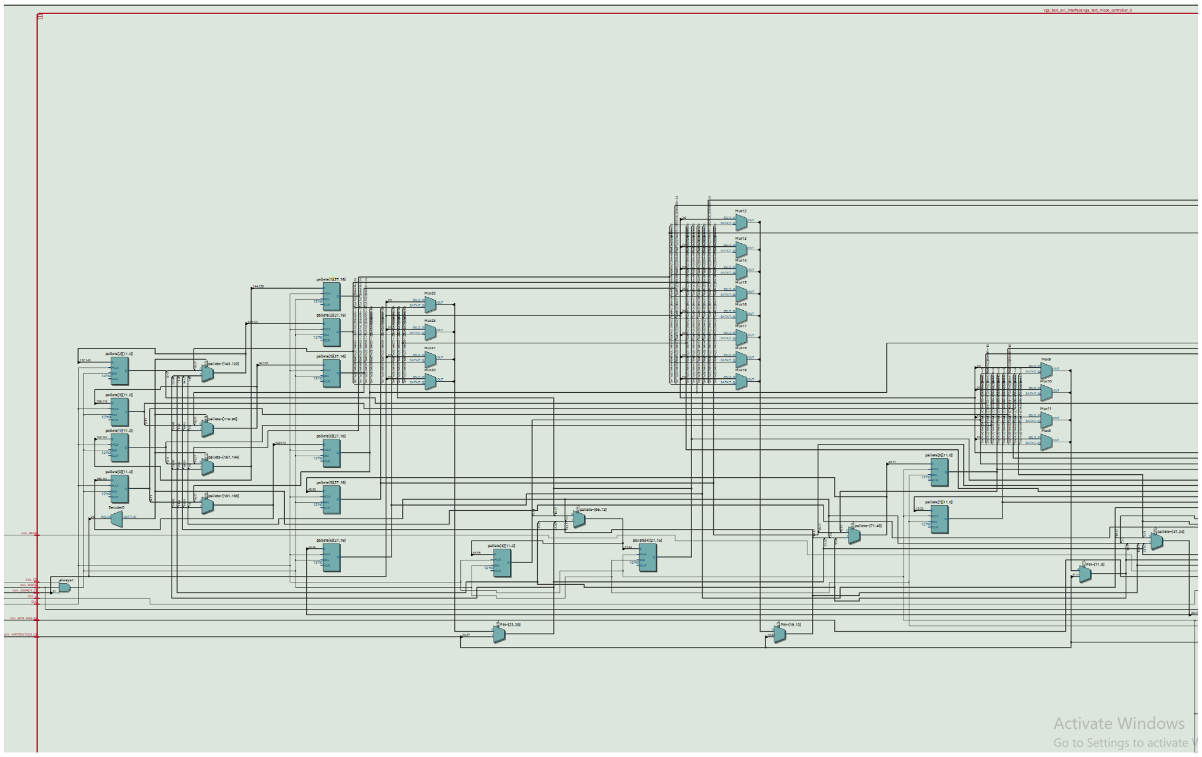
Note: The above diagram is one that has the registers combined into 1 register. This is to show the other aspects without the registers making it unseeable.

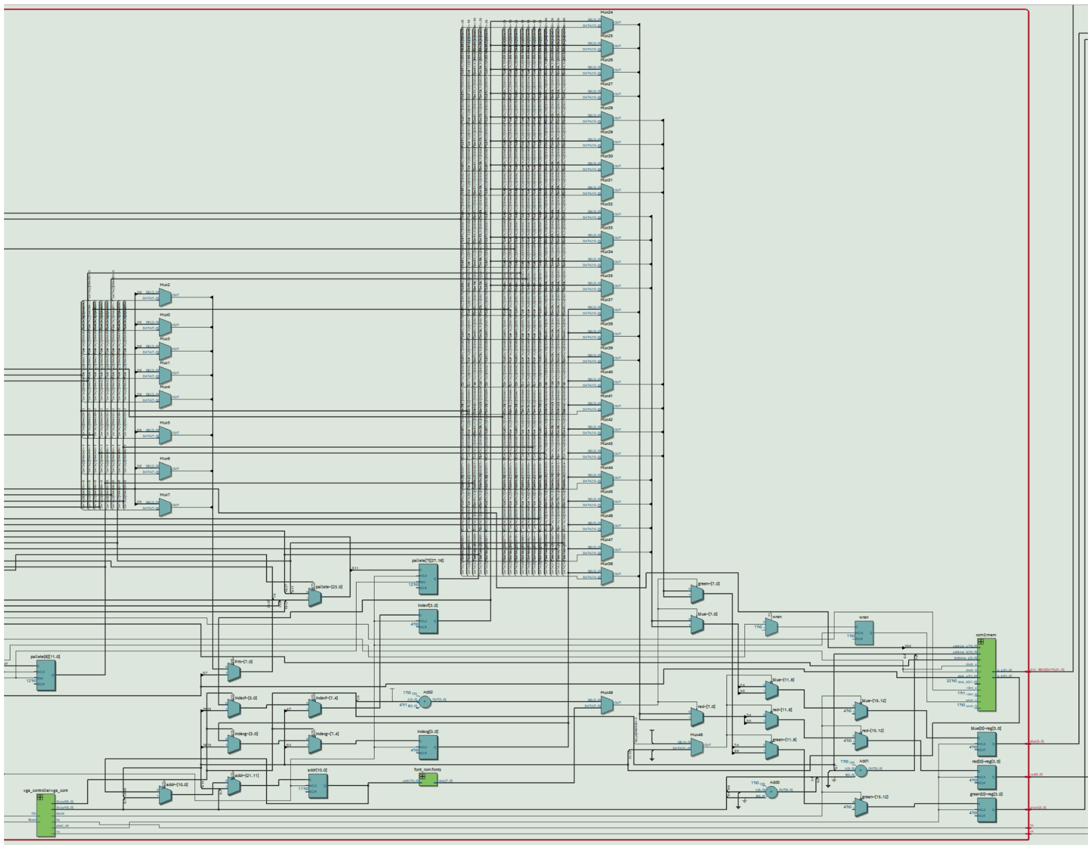
Week 2 Top Level



Week 2 vga_text_avl_interface.sv Block Diagram (Second and Third are zooms)







Written Description of .v and .sv Modules

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.

Module: **lab7** in lab7.sv

Inputs: MAX10_CLK1_50, KEY[1:0], SW[9:0]

Outputs: LEDR[9:0], HEX0[7:0], HEX1[7:0], HEX2[7:0], HEX3[7:0], HEX4[7:0], HEX5[7:0], DRAM_CLK, DRAM_CKE, DRAM_ADDR[12:0], DRAM_BA[1:0], DRAM_DQ[15:0], DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, VGA_R[3:0], VGA_G[3:0], VGA_B[3:0]

In/Outs: ARDUINO_IO[15:0], ARDUINO_RESET_N

Description: The top level module takes in signals such as the clock, switch inputs, and button inputs and is able to output signals to the hex driver and the LED strip, as well as different signals for the SDRAM and the VGA RGB values and the horizontal sync and vertical sync signals. The main functionality of this module is instantiating the lab71soc or lab72soc modules, which correspond to the Verilog HDL code for the Lab 7.1 or Lab 7.2 SoCs, respectively.

Purpose: This is the top level module used for both Lab 7.1 and 7.2.

Module: **vga_controller** in VGA_controller.sv

Inputs: Clk, Reset

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

Description: This module is the controller which controls the behavior of the VGA port and provides signals to the monitor and the rest of the modules. This module has two counters, a vertical and horizontal one which provide a HS and VS signal to sync the monitor with the FPGA. The module also provides a DrawX and Y signal which is the position of the current pixel being drawn. A blank signal is also provided to show nothing should be drawn. The pixel_clock was not used in this lab, as well as the sync signal.

Purpose: Module used to control and Sync the VGA port of the FPGA with the monitor.

Module: **vga_text_avl_interface** in vga_text_avl_interface.sv

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

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

Description: For the first week, we used this module to instantiate 601 thirty-two bit registers, but in the second week, we instantiated the ocm2 module which is our on-chip memory. The module also instantiates the font_rom and vga_controller modules. We included some logic in the module that allows us to read and write to and from the Avalon memory mapped bus and the VRAM. The module uses the frame buffer/ memory to draw the screen.

Purpose: This module is used to interface with the CPU and the VRAM, which has two different implementations for the two weeks we've worked on the lab. The inputs are signals from the avalon memory mapped slave, which are taken and used to output RGB values as well as the character that is being read/written.

Module: **font_rom** in font_rom.sv

Inputs: addr[10:0]

Outputs: data[7:0]

Description: This module is a ROM that holds the data for all the 128 different characters/glyphs to be drawn. Each glyph has 16 lines that have 8 bits of data each which correspond to whether the pixel should be lit up or not. The address is found by doing 16*index+Y position. The resultant data is then drawn from left to right which needs a 8-DrawX%8 due to the endian discrepancy.

Purpose: This module allows us to access different characters so that we are able to draw them on the screen.

Module: **HexDriver** in HexDriver.sv

Inputs: In0[3:0]

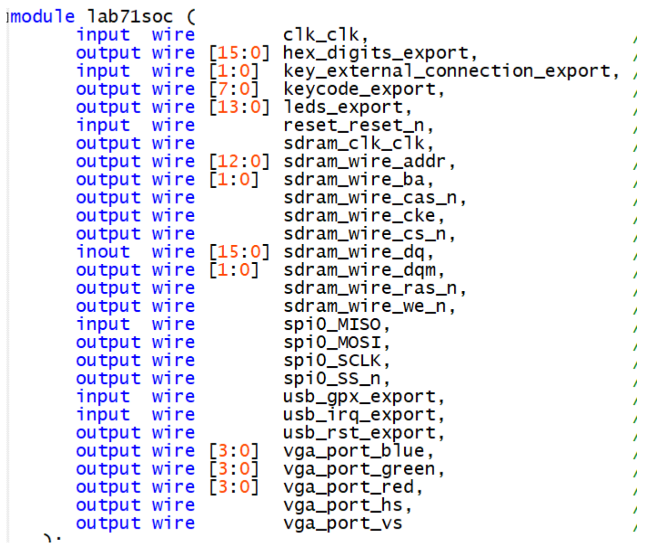
Outputs: Out0[6:0]

Description: HexDriver is used to map the inputs to outputs that can be seen on the hex display.

Purpose: Maps the input to outputs that can be displayed on the HEX display.

Module: **lab71soc** in lab71soc.v

Inputs/Outputs:



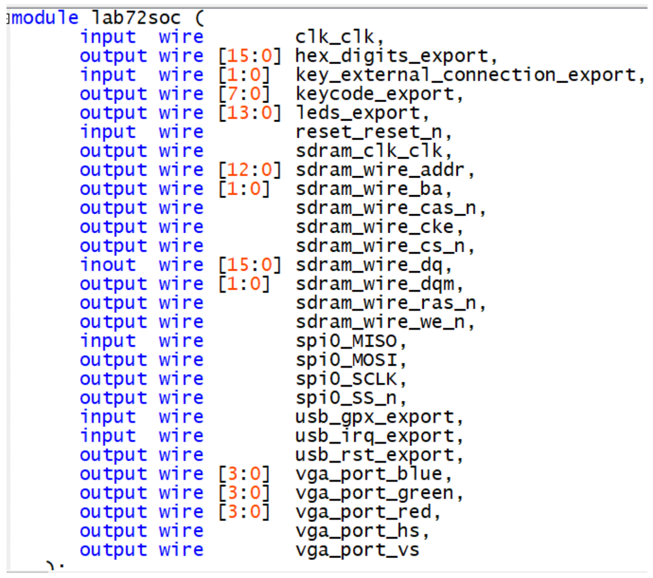
```
module lab71soc (
      input  wire        clk_clk,
      output wire [15:0] hex_digits_export,
      input  wire [1:0]  key_external_connection_export,
      output wire [7:0]  keycode_export,
      output wire [13:0] leds_export,
      input  wire        reset_reset_n,
      output wire        sdram_clk_clk,
      output wire [12:0] sdram_wire_addr,
      output wire [1:0]  sdram_wire_ba,
      output wire        sdram_wire_cas_n,
      output wire        sdram_wire_cke,
      output wire        sdram_wire_cs_n,
      inout  wire [15:0] sdram_wire_dq,
      output wire [1:0]  sdram_wire_dqm,
      output wire        sdram_wire_ras_n,
      output wire        sdram_wire_we_n,
      input  wire        spi0_MISO,
      output wire        spi0_MOSI,
      output wire        spi0_SCLK,
      output wire        spi0_SS_n,
      input  wire        usb_gpx_export,
      input  wire        usb_irq_export,
      output wire        usb_rst_export,
      output wire [3:0]  vga_port_blue,
      output wire [3:0]  vga_port_green,
      output wire [3:0]  vga_port_red,
      output wire        vga_port_hs,
      output wire        vga_port_vs
  );
```

Description: This is the HDL version of the SoC we developed in Platform Designer for Lab 7.1. Screenshots of the IP blocks are provided in the System Level Block Diagram section.

Purpose: We instantiate this SoC module in lab7.sv.

Module: **lab72soc** in lab72soc.v

Inputs/Outputs:



```
module lab72soc (
      input  wire        clk_clk,
      output wire [15:0] hex_digits_export,
      input  wire [1:0]  key_external_connection_export,
      output wire [7:0]  keycode_export,
      output wire [13:0] leds_export,
      input  wire        reset_reset_n,
      output wire        sdram_clk_clk,
      output wire [12:0] sdram_wire_addr,
      output wire [1:0]  sdram_wire_ba,
      output wire        sdram_wire_cas_n,
      output wire        sdram_wire_cke,
      output wire        sdram_wire_cs_n,
      inout  wire [15:0] sdram_wire_dq,
      output wire [1:0]  sdram_wire_dqm,
      output wire        sdram_wire_ras_n,
      output wire        sdram_wire_we_n,
      input  wire        spi0_MISO,
      output wire        spi0_MOSI,
      output wire        spi0_SCLK,
      output wire        spi0_SS_n,
      input  wire        usb_gpx_export,
      input  wire        usb_irq_export,
      output wire        usb_rst_export,
      output wire [3:0]  vga_port_blue,
      output wire [3:0]  vga_port_green,
      output wire [3:0]  vga_port_red,
      output wire        vga_port_hs,
      output wire        vga_port_vs
  );
```

Description: This is the HDL version of the SoC we developed in Platform Designer for Lab 7.2. Screenshots of the IP blocks are provided in the System Level Block Diagram section.

Purpose: We instantiate this SoC module in lab7.sv.

Module: **ocm2** in ocm2.v

Inputs: address_a[10:0], address_b[10:0], clock_a, clock_b, data_a[31:0], data_b[31:0], rden_a, rden_b, wren_a, wren_b

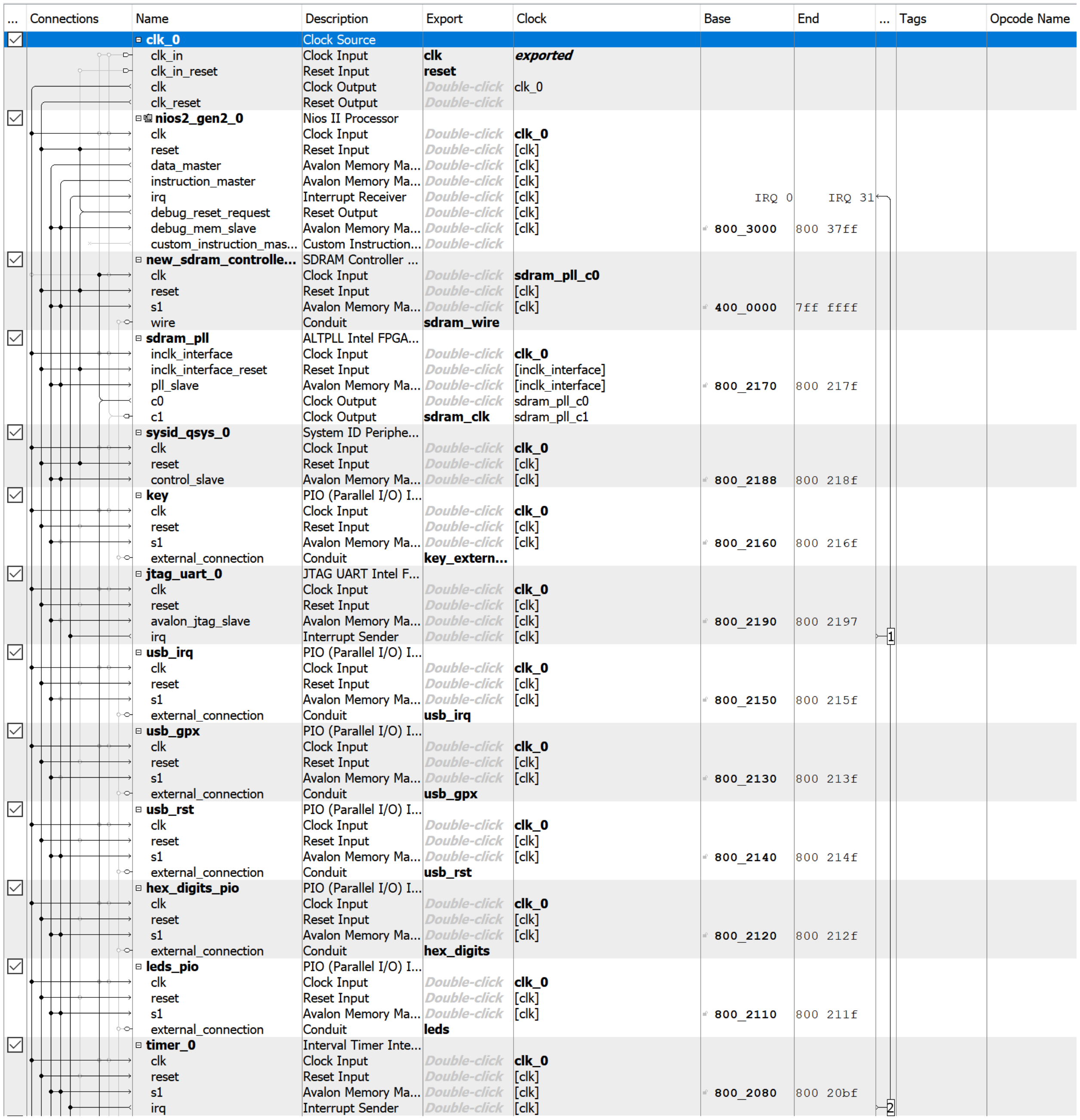
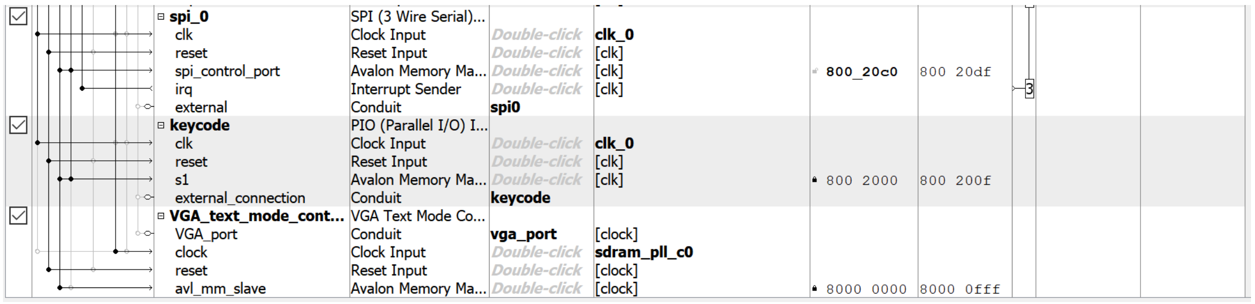
Outputs: q_a[31:0], q_b[31:0]

Description: We used the IP Catalog to create the OCM. This ocm has two inputs and two outputs, each with its own clock, read enable, and write enable signal. This OCM holds 1200 words of memory where each word is 32 bits. The OCM is run at 200mhz.

Purpose: This module is instantiated as the on-chip memory and is used as an implementation of VRAM for Lab 7.2.

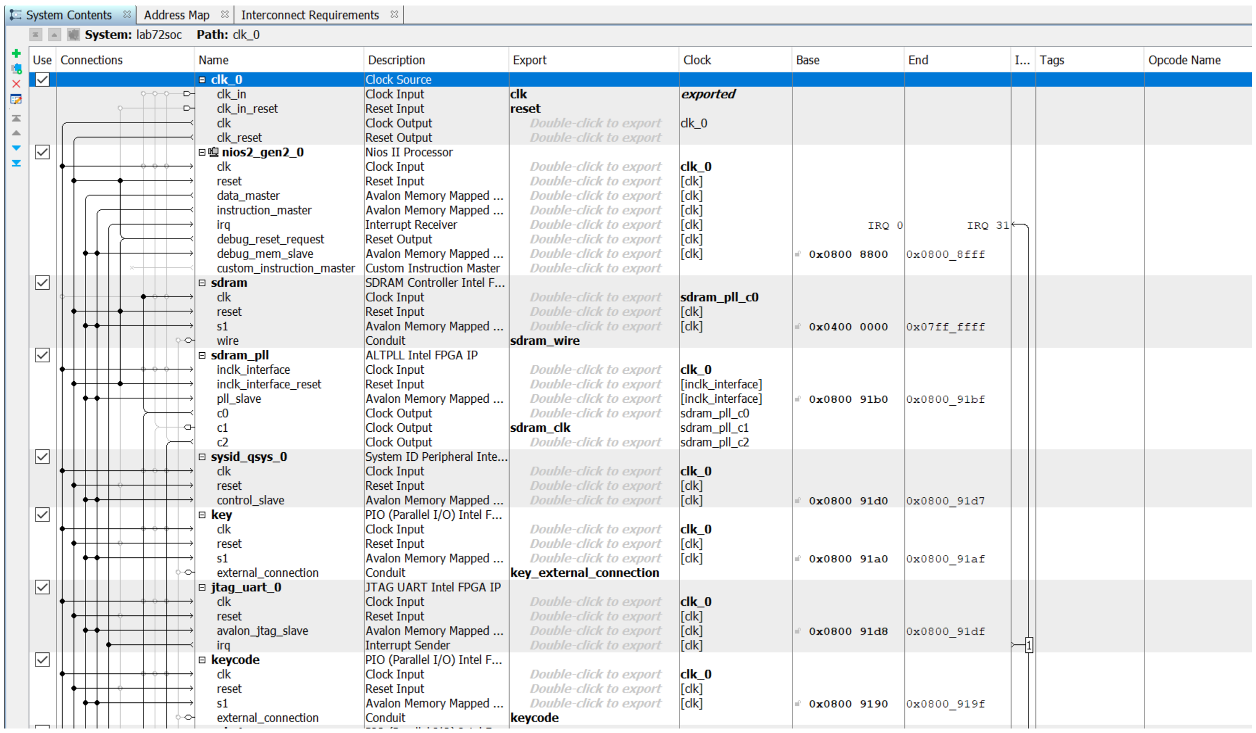
System Level Block Diagram

Lab 7.1 Platform Designer View:

...	Connections	Name	Description	Export	Clock	Base	End	...	Tags	Opcode Name
✓		⊟ **clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-click*	clk_0					
		clk_reset	Reset Output	*Double-click*						
✓		⊟ **nios2_gen2_0**	Nios II Processor							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		data_master	Avalon Memory Ma...	*Double-click*	[clk]					
		instruction_master	Avalon Memory Ma...	*Double-click*	[clk]					
		irq	Interrupt Receiver	*Double-click*	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	*Double-click*	[clk]					
		debug_mem_slave	Avalon Memory Ma...	*Double-click*	[clk]	800_3000	800 37ff			
		custom_instruction_mas...	Custom Instruction...	*Double-click*						
✓		⊟ **new_sdram_controlle...**	SDRAM Controller ...							
		clk	Clock Input	*Double-click*	**sdram_pll_c0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	400_0000	7ff ffff			
		wire	Conduit	**sdram_wire**						
✓		⊟ **sdram_pll**	ALTPLL Intel FPGA...							
		inclk_interface	Clock Input	*Double-click*	**clk_0**					
		inclk_interface_reset	Reset Input	*Double-click*	[inclk_interface]					
		pll_slave	Avalon Memory Ma...	*Double-click*	[inclk_interface]	800_2170	800 217f			
		c0	Clock Output	*Double-click*	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
✓		⊟ **sysid_qsys_0**	System ID Periphe...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		control_slave	Avalon Memory Ma...	*Double-click*	[clk]	800_2188	800 218f			
✓		⊟ **key**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2160	800 216f			
		external_connection	Conduit	**key_extern...**						
✓		⊟ **jtag_uart_0**	JTAG UART Intel F...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		avalon_jtag_slave	Avalon Memory Ma...	*Double-click*	[clk]	800_2190	800 2197			
		irq	Interrupt Sender	*Double-click*	[clk]			1		
✓		⊟ **usb_irq**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2150	800 215f			
		external_connection	Conduit	**usb_irq**						
✓		⊟ **usb_gpx**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2130	800 213f			
		external_connection	Conduit	**usb_gpx**						
✓		⊟ **usb_rst**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2140	800 214f			
		external_connection	Conduit	**usb_rst**						
✓		⊟ **hex_digits_pio**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2120	800 212f			
		external_connection	Conduit	**hex_digits**						
✓		⊟ **leds_pio**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2110	800 211f			
		external_connection	Conduit	**leds**						
✓		⊟ **timer_0**	Interval Timer Inte...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800_2080	800 20bf			
		irq	Interrupt Sender	*Double-click*	[clk]			2		
✓		⊟ **spi_0**	SPI (3 Wire Serial)...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		spi_control_port	Avalon Memory Ma...	*Double-click*	[clk]	800_20c0	800 20df			
		irq	Interrupt Sender	*Double-click*	[clk]			3		
		external	Conduit	**spi0**						
✓		⊟ **keycode**	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800 2000	800 200f			
		external_connection	Conduit	**keycode**						
✓		⊟ **VGA_text_mode_cont...**	VGA Text Mode Co...							
		VGA_port	Conduit	**vga_port**	[clock]					
		clock	Clock Input	*Double-click*	**sdram_pll_c0**					
		reset	Reset Input	*Double-click*	[clock]					
		avl_mm_slave	Avalon Memory Ma...	*Double-click*	[clock]	8000 0000	8000 0fff			

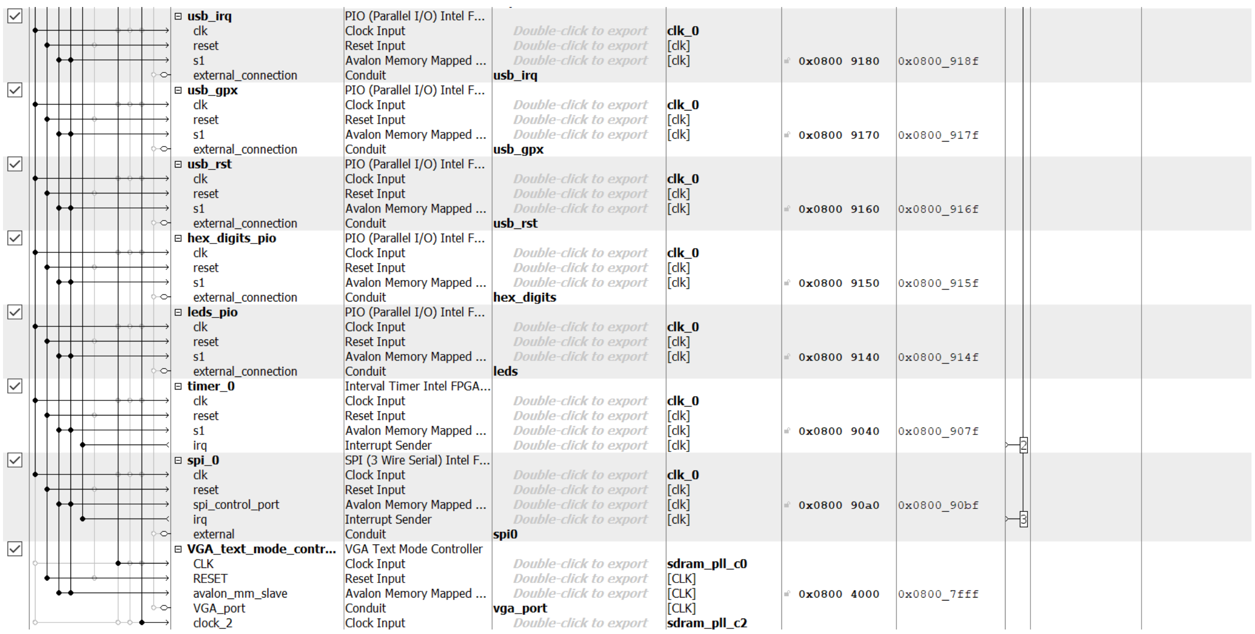
Lab 7.2 Platform Designer View:



System Contents | Address Map | Interconnect Requirements

System: lab72soc **Path:** clk_0

Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags	Opcode Name
✓		**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 ...	*Double-click to export*	[clk]					
		instruction_master	Avalon Memory Mapped ...	*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 ...	*Double-click to export*	[clk]	0x0800 8800	0x0800_8fff			
		custom_instruction_master	Custom Instruction Master	*Double-click to export*						
✓		**sdram**	SDRAM Controller Intel F...							
		clk	Clock Input	*Double-click to export*	**sdram_pll_c0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*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_interface]					
		pll_slave	Avalon Memory Mapped ...	*Double-click to export*	[inclk_interface]	0x0800 91b0	0x0800_91bf			
		c0	Clock Output	*Double-click to export*	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
		c2	Clock Output	*Double-click to export*	sdram_pll_c2					
✓		**sysid_qsys_0**	System ID Peripheral Inte...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		control_slave	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 91d0	0x0800_91d7			
✓		**key**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 91a0	0x0800_91af			
		external_connection	Conduit	**key_external_connection**						
✓		**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 ...	*Double-click to export*	[clk]	0x0800 91d8	0x0800_91df			
		irq	Interrupt Sender	*Double-click to export*	[clk]			1		
✓		**keycode**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9190	0x0800_919f			
		external_connection	Conduit	**keycode**						



Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags	Opcode Name
✓		**usb_irq**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9180	0x0800_918f			
		external_connection	Conduit	**usb_irq**						
✓		**usb_gpx**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9170	0x0800_917f			
		external_connection	Conduit	**usb_gpx**						
✓		**usb_rst**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9160	0x0800_916f			
		external_connection	Conduit	**usb_rst**						
✓		**hex_digits_pio**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9150	0x0800_915f			
		external_connection	Conduit	**hex_digits**						
✓		**leds_pio**	PIO (Parallel I/O) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9140	0x0800_914f			
		external_connection	Conduit	**leds**						
✓		**timer_0**	Interval Timer Intel FPGA...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 9040	0x0800_907f			
		irq	Interrupt Sender	*Double-click to export*	[clk]			2		
✓		**spi_0**	SPI (3 Wire Serial) Intel F...							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		spi_control_port	Avalon Memory Mapped ...	*Double-click to export*	[clk]	0x0800 90a0	0x0800_90bf			
		irq	Interrupt Sender	*Double-click to export*	[clk]			3		
		external	Conduit	**spi0**						
✓		**VGA_text_mode_contr...**	VGA Text Mode Controller							
		CLK	Clock Input	*Double-click to export*	**sdram_pll_c0**					
		RESET	Reset Input	*Double-click to export*	[CLK]					
		avalon_mm_slave	Avalon Memory Mapped ...	*Double-click to export*	[CLK]	0x0800 4000	0x0800_7fff			
		VGA_port	Conduit	**vga_port**	[CLK]					
		clock_2	Clock Input	*Double-click to export*	**sdram_pll_c2**					

Functionality of IP Blocks

IP Block: **clk_0**
Lab 7.1 or Lab 7.2: Both
Functionality: This is the IP block that contains a clock output for the rest of the modules. It is set to 50 MHz. Its input comes from the FPGA oscillator.

IP Block: **nios2_gen2_0** (NIOS II Processor)
Lab 7.1 or Lab 7.2: Both
Functionality: This is the CPU that is used in this lab. It is in charge of the accumulation circuit and the USB/SPI communication. Main CPU where the C code runs on. Also used to access different parts of the Avalon Bus through the C code.

IP Block: **sdram**
Lab 7.1 or Lab 7.2: Both
Functionality: Memory used to store information for the CPU. Maps to the SDRAM on the FPGA. Used as high capacity memory for the CPU to store and run programs.

IP Block: **sdram_pll**
Lab 7.1 or Lab 7.2: Both
Functionality: This module is used to create a clock signal that is out of phase from the main clock at a specific angle. This is used to drive the SDRAM controller and the FPGA SDRAM at different clocks. This skew is needed due to the precise timing of the SDRAM and is used to compensate for the delay and inaccuracies between the SDRAM and the controller. This was also used to drive the OCM at 200mhz in our implementation to reduce lag/glitches.

IP Block: **sysid_qsys_0**
Lab 7.1 or Lab 7.2: Both
Functionality: This is used to ensure compatibility between the hardware and software components, for example the NIOS system and the loaded program on the FPGA. Gives a warning when there is a mismatch of the serial number provided. Used for warning and debugging purposes.

IP Block: **key**
Lab 7.1 or Lab 7.2: Both (Carried over from previous labs)
Functionality: This module is a PIO, a parallel IO module. Specifically this is a 2-bit input module that is mapped to the buttons on the FPGA. This is used for the accumulation circuit, where run is needed to be pressed to see when to accumulate, and reset in some other cases.

IP Block: **jtag_uart**
Lab 7.1 or Lab 6.2: Both
Functionality: This module is used to communicate with the NIOS system while it's running through the computer terminal. This allows statements such as print and scan to work in C, and is used mainly for debugging.

IP Block: **keycode**
Lab 7.1 or Lab 7.2: Both (Carried over from previous labs)
Functionality: This is a PIO block. This is a 8-bit output port that is a part of the SOC. This is used to export the read character from the keyboard to the top-level sv module which then goes

into the ball module.
IP Block: **usb_irq** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is a 1-bit PIO block. It is in input mode. This is used as an interrupt port that the MAX chip uses. Used for interrupt functionality in some cases. This is mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: **usb_gpx** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is a 1-bit input PIO block. It is required for the MAX chip, and is a general purpose push pull output. Functionality is decided by the chip and depends on different factors. This is mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: **usb_rst** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is a 1-bit output PIO block. It is required for the MAX chip. This is mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: **hex_digits_pio** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is a 16-bit output PIO block. This is used to output the hex of the keycode. Used to map the FPGA ports to the SOC in the top-level to display on the HEX display. The NIOS system sets the output according to the current keyboard presses.
IP Block: **leds_pio** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is a 14-bit output PIO block. This is used to display the LED accumulation on the FPGA leds. Mapped to the ports of the FPGA from the top-level SOC.
IP Block: **timer_0** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is a timer that keeps track of the timeouts that the USB requires. This is required for the USB SPI implementation on the MAX chip. An interrupt is assigned which is connected to the NIOS system.
IP Block: **spi_0** Lab 7.1 or Lab 7.2: Both (Carried over from previous labs) Functionality: This is the SPI port peripheral that is required to use SPI between the NIOS systema and the MAX chip. In this case the NIOS system is the host and MAX is the slave. The SPI ports are connected via the top-level to the MAX chip via the FPGA.
IP Block: **VGA_text_mode_controller** Lab 7.1 or Lab 7.2: Both Functionality: This is a custom IP core that was developed to address the requirement of having an interface in between the CPU and VRAM. To address this, a set of signals through an Avalon Memory Mapped Slave are used. The VGA port allows for the processing of the

RGB signals as well as the vsync and hsync signals. This IP stores the VRAM either in register or OCM format, and instantiates the font_rom and VGA controller. It then uses this to draw the screen according to the current VRAM and can either be palletized or not. This module allows the NIOS system to write to the VRAM, read from it, and set the palette to allow the screen saver and other animations to run with different concurrent colors.

Post Lab Questions

1. Refer to the Design Resources and Statistics in IQT.17-19 and complete the following design statistics table

Lab 7.1

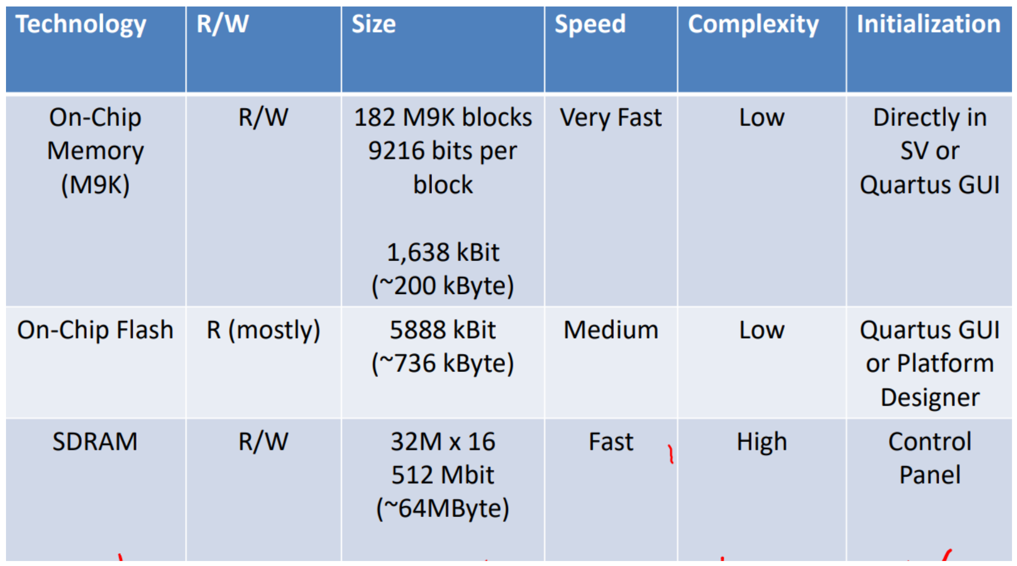
LUT	37,331
DSP	0
BRAM	11,264
Flip-Flop	22,016
Frequency Mhz	57.15
Static Power mW	97.43 mW
Dynamic Power mW	253.27 mW
Total Power mW	372.76 mW

Lab 7.2

LUT	5247
DSP	0
BRAM	49664
Flip-Flop	2957
Frequency Mhz	72.61
Static Power mW	96.72 mW
Dynamic Power mW	103.77 mW
Total Power mW	222.55 mW

2. Discuss the difference between using on-chip memory for VRAM and registers. Which design is more efficient, what are the tradeoffs?

The cleaner aspect of the on-chip memory is that it only requires 2 inputs and 2 outputs while there are 601 inputs and outputs for the registers. In terms of design and efficiency, the on chip memory takes less LUTs to implement while the 600 registers require a lot more. However, the tradeoff is that the on-chip memory requires more absolute memory than the registers. However, the biggest noticeable difference in terms of efficiency is the compilation time. To compile the 601 registers, we were regularly seeing compilation times of over 10 minutes. For the on-chip memory, the compilation time was reduced to a much better time of under 3-4 minutes. The registers are also beneficial in that they can be accessed in parallel which makes it easier to draw, while the OCM only has 2-ports which limits the ways to draw. From the data above the OCM seems to be much more efficient, while a bit more complex in design but it run faster and uses less power and less space.



Technology	R/W	Size	Speed	Complexity	Initialization
On-Chip Memory (M9K)	R/W	182 M9K blocks 9216 bits per block 1,638 kBit (~200 kByte)	Very Fast	Low	Directly in SV or Quartus GUI
On-Chip Flash	R (mostly)	5888 kBit (~736 kByte)	Medium	Low	Quartus GUI or Platform Designer
SDRAM	R/W	32M x 16 512 Mbit (~64MByte)	Fast	High	Control Panel

Conclusion

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

Our design functioned almost perfectly. All the parts worked as expected, the only thing that could be improved is the efficiency of the design such as by reducing the number of if statements and putting as many things in combination logic as possible.

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

This lab is very useful for our final project. This lab will allow us to draw sprites instead of glyphs to draw sprites to be able to design our game. This also taught us how to use OCM as a frame buffer and how we can use that to draw. Some extensions include using sprites instead of glyphs, and increasing the number of colors.

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.

No.

Overall, this lab was a good path forward from Lab 6, and a good way to end the series of labs that we are required to do, because of how it is testing us on how well we've understood the concepts from previous labs and then some more concepts until now. It also sets us in a good direction to be working on the Final Project, because we were introduced to palletizing the RGB values for a sprite, which is important for students trying to do a graphical final project. That ended up being something that was important for our final project, because it taught us another way to implement graphics. The thing that helped us the most in order to implement the lab was the lecture that explained how the VGA text mode controller worked, because the diagrams made the concepts easier to understand.