ECE 385

Spring 2023

Experiment #7

VGA Text Mode Controller with Avalon-MM Interface

Dylan Bautista, Matthew Wei

Lab Section NL/ April 16 11:00 AM

Nick Lu

Experiment #7

VGA Text Mode Controller with Avalon-MM Interface

I. Introduction/Purpose of Circuit:

This design is intended to create a text mode graphics controller, initially with monochrome text display, and in week 2 with color text display. The goal is to display up to 30 rows of 80 individual 8x16 pixel characters on an VGA display monitor output, either inverted or standard colors. The input to the display is through a VRAM which can be accessed to be read or written through the Avalon interface. Thereby, video driver software code will be able to affect what is displayed on screen. Additionally, there is either a single control register in VRAM or a color palette which can alter the color output.

This lab builds off of the SOC design of lab6.2, in which we defined a top-level System Verilog entity where the VGA_controller and Color_mapper is instantiated. During this lab 7, we went more in depth with VGA sprite drawing in order to transfer the information from the pixel-row-wise Font_ROM to the VGA drawing modules.

II. Written description of Circuit

Week 1 (Monochrome Text Display)

The lab 7.1 system encompassed a VRAM of 600 32-bit System-Verilog registers that can each by individually accessed, in addition to the control register. This VRAM is controlled within our VGA_text_interface file, along with the Font_ROM, VGA controller, and exported signals. The Font_ROM, which is supplied to us, stores each glyph's pixel information by row. The VGA_text_interface serves as the top-level file for the core component on platform designer

and houses the logic for writing and reading via the Avalon Bus signals, along with the color_mapper/VGA_controller signals.

The VGA_text_mode component was added to our platform designer layout along with the accompanying vga_text_avl_interface.sv file. The IP core is aimed to be reusable hardware with specified inputs and outputs built solely towards the production of text-mode graphics. The .sv file allows us to interact with the signals from the NIOSII processor via the Avalon slave signals. There is a read, write, readdata, writedata, address, byteneable, and chipselect I/O sequence within the interface hardware. Our code initializes a 32-bit local register base of 601 spaces, then uses conditional statements to check for AVL signals. In the case of write, the specified bits of writedata are written into the registers. In the case of read, the corresponding address's register output is stored in readdata.

To get the correct location for the glyph on screen, we first divide DrawX by 8 and DrawY by 16 to get the character row and column respectively. Then, the position of the VRAM address is calculated by getting [(row*80) +col]/4. Depending on what the last two bits of the column is, the top 7 bits of the sprite_address are set from a section of the VRAM data, with the remaining 4 being the last 4 bits of DrawY to get the row of the specific glyph. Additionally, the inverse bit is set from the 8^{th} bit of the section of VRAM data. The sprite_address is sent into the Font_ROM and the data sent out is the 8 bits of that row. The specific pixel from the output data is found using index 7-DrawX [2:0]. Using the pixel clock, the pixel signal is XORed with the invert signal to determine if the overall RGB output values should be from the foreground slots or the background slots within the control register.

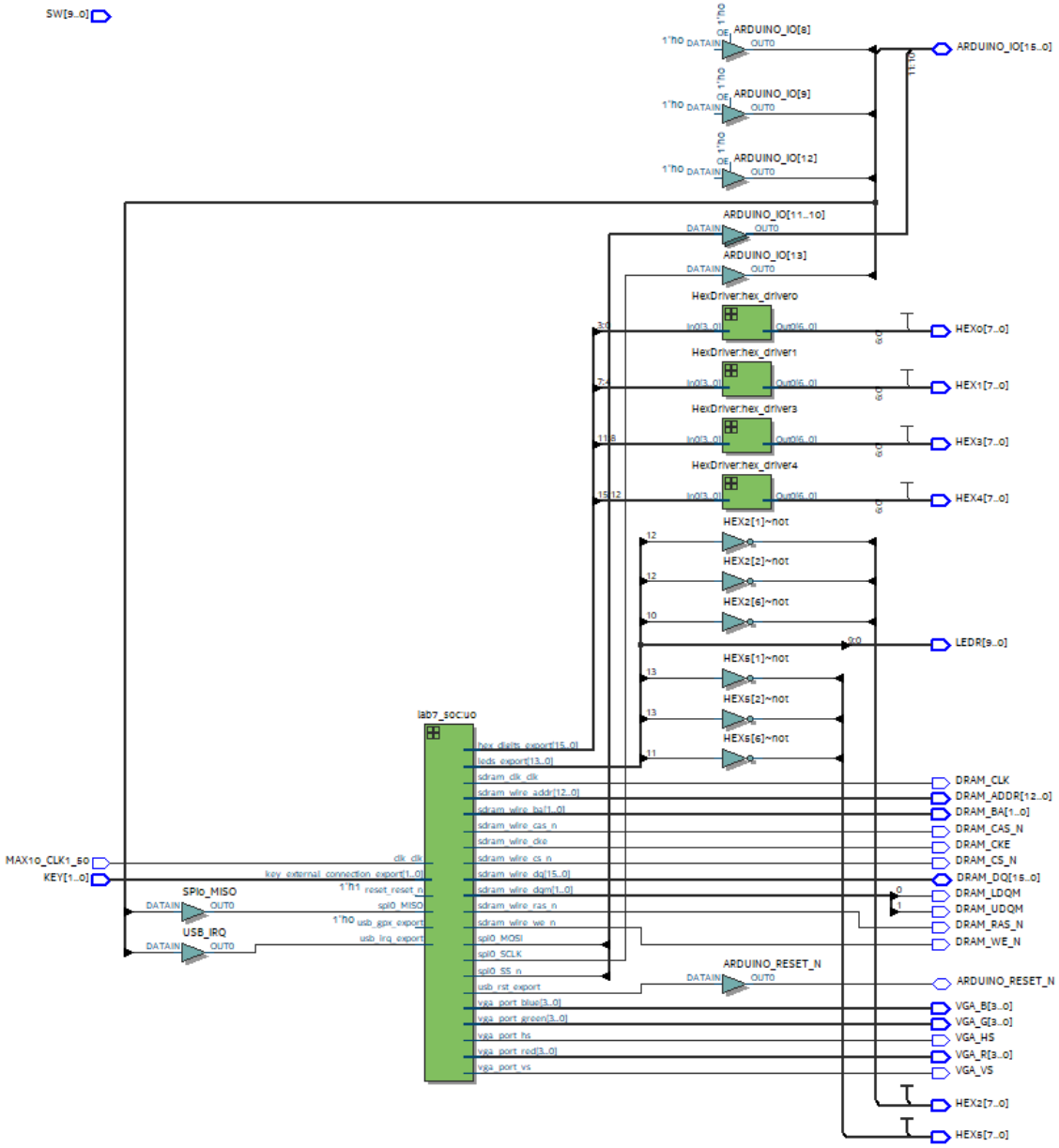
Week 2 (Color Text Display)

The main difference in week 2's design was that the introduction of character-based color required that each 32-bit VRAM encoding could only hold 2 characters. Therefore, the VRAM doubles in size, requiring on-chip memory usage to accommodate. In order to do this, we instantiated an on-chip memory megafunction, creating a RAM module which can be instantiated within the VGA_text_interface file to hold up to 2048 words of memory. Because the ram module only has two ports that can be accessed at a time, we used one for Avalon reading and writing, and the other one solely for reading the VRAM information via the logic in our code.

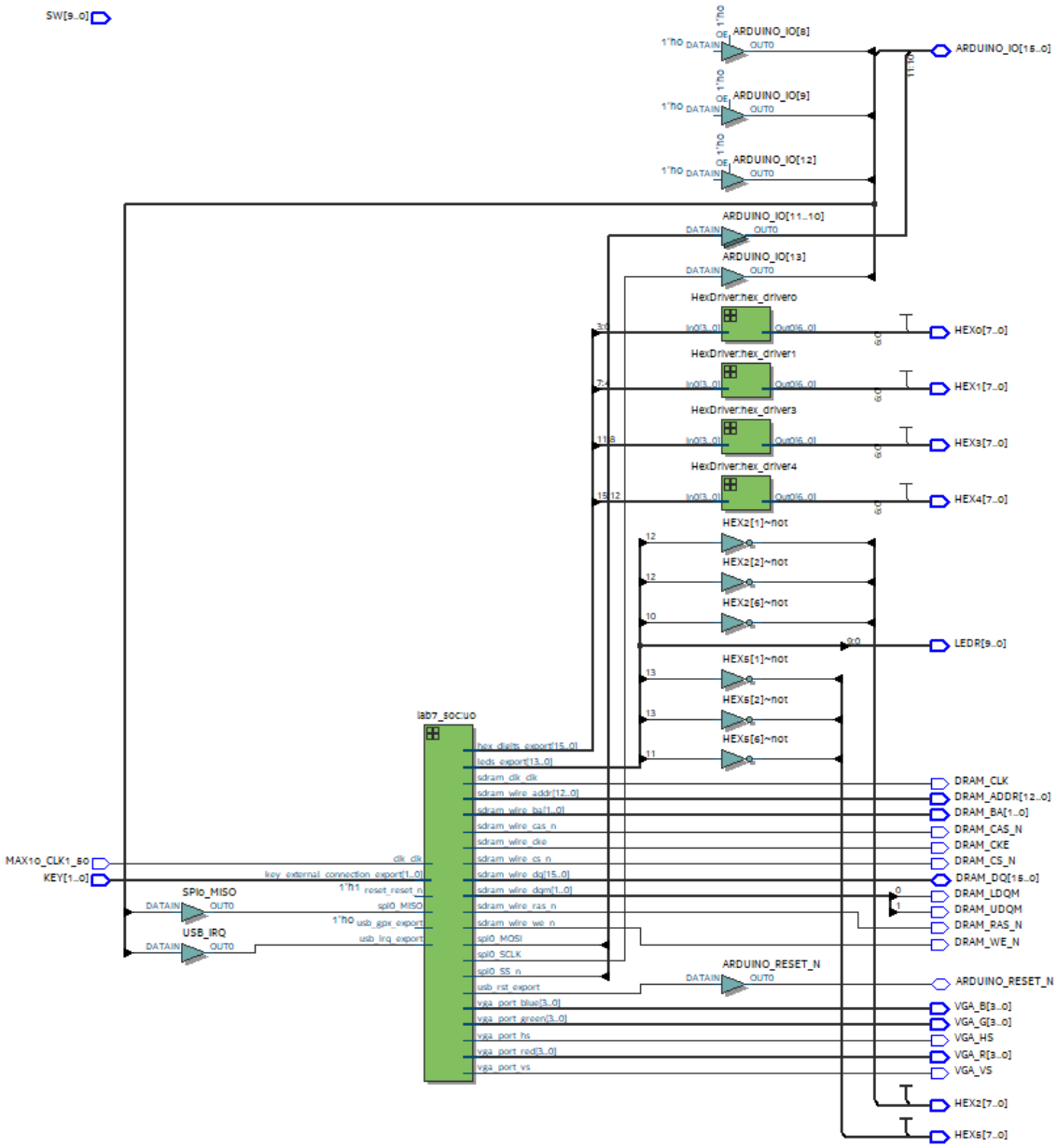
The only difference in the platform designer is that the address width for the VGA_text_interface needs to increase from 10 to 12 to accommodate the doubling VRAM and the external color palette. In terms of hardware, the vram_address that is calculated from the pixel location now comes from[(row*80) +col]/2 instead of dividing by 4 since there's only 2 characters per word. Additionally, the top 7 bits of the sprite_address come from one of 2 different sections of VRAM data rather than 4. The invert bit, foreground index, and background index additionally are set from certain bits of this section. The hardware stays consistent until the RGB values are set. For multicolored text, eight 32-bit words are reserved after constant reserved space. This palette is organized through C software, which sets a bit mask to the corresponding portion of the palette word, then OR'ed with the red, green, and blue inputs at their corresponding positions. The correct foreground and background RGB values are selected from the palette register using the stored indices, using the bottom bits to determine if it's an odd-or even character. The invert and pixel values are XOR'ed to determine if the foreground or background RGB values should be stored into the overall RGB outputs on the pixel clock edge.

III. Block Diagram:

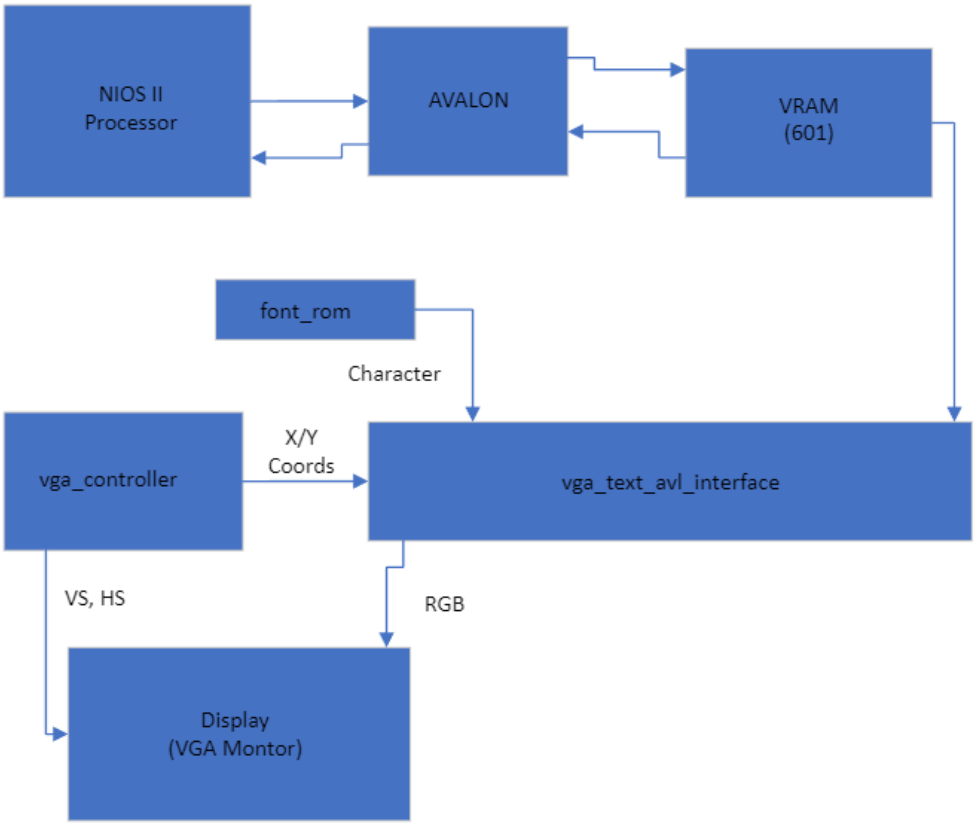
Top Level Diagram



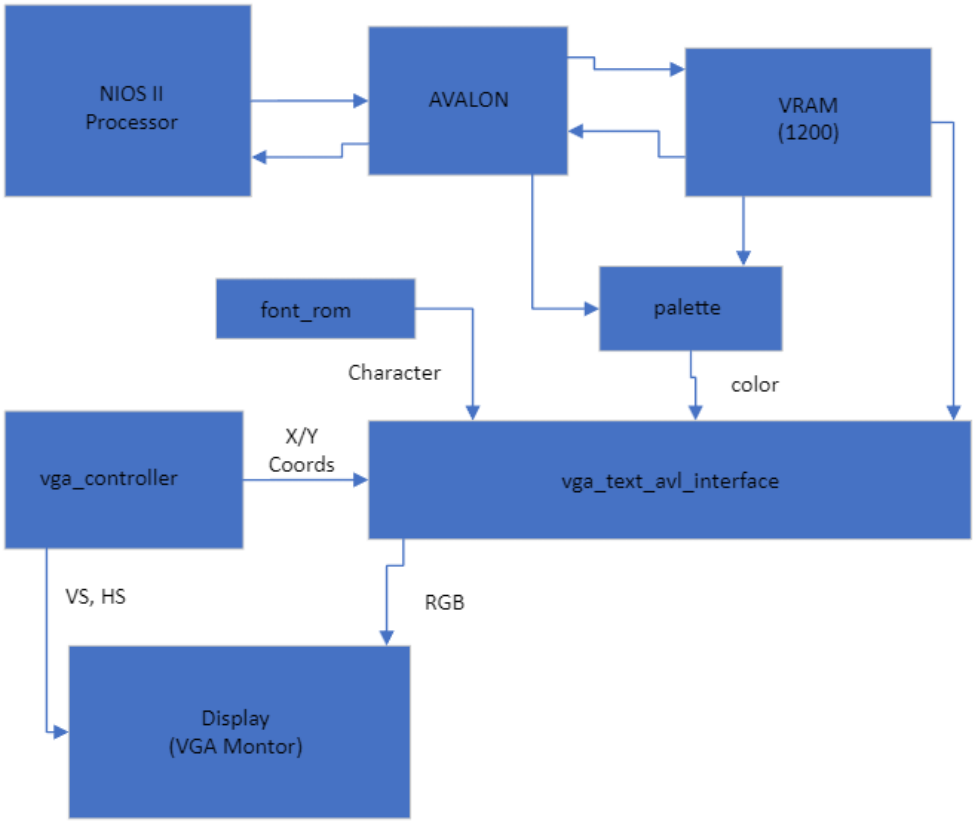
Lab 7.1 RTL Block Diagram



Lab 7.2 RTL Block Diagram

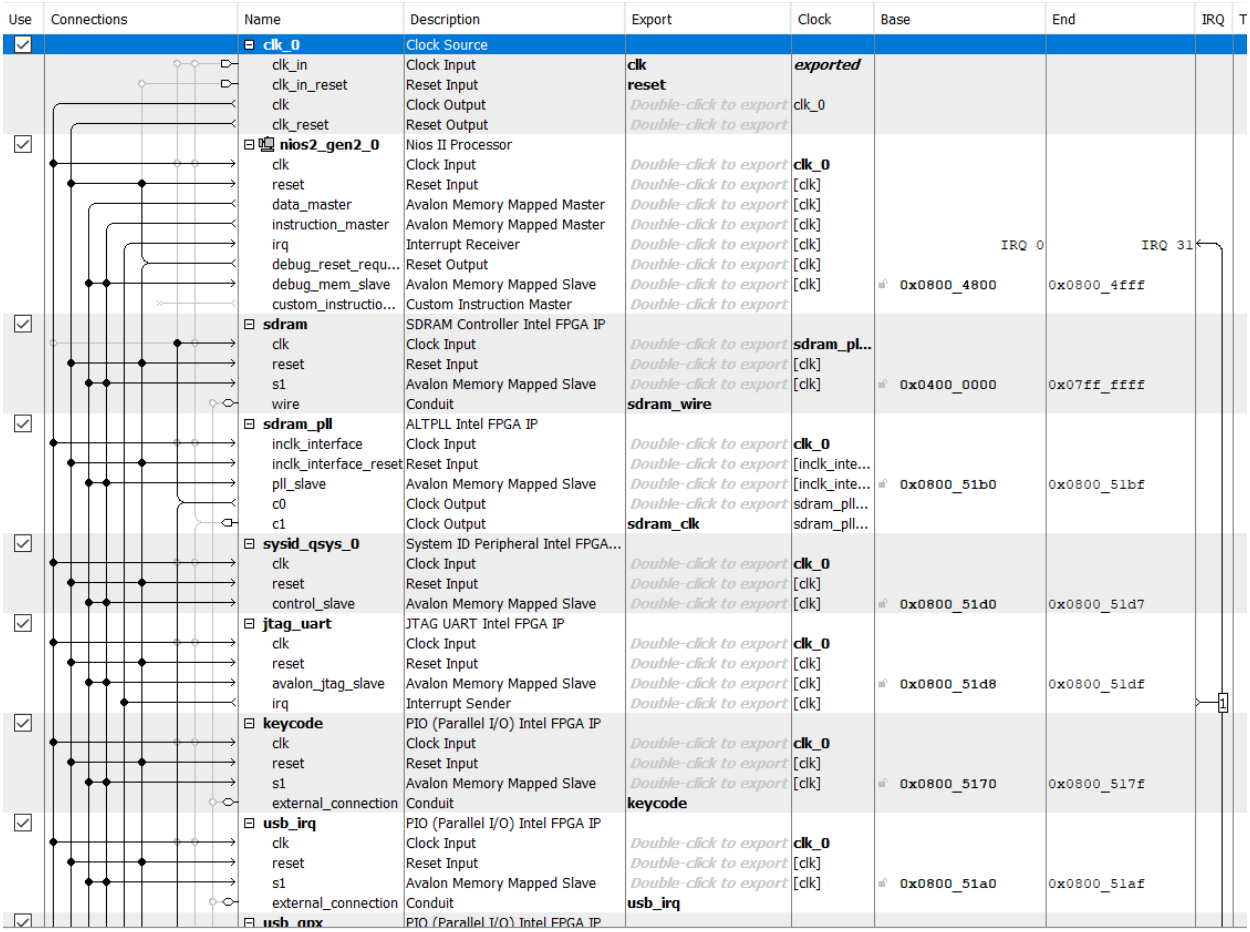


Lab 7.1 Block Diagram

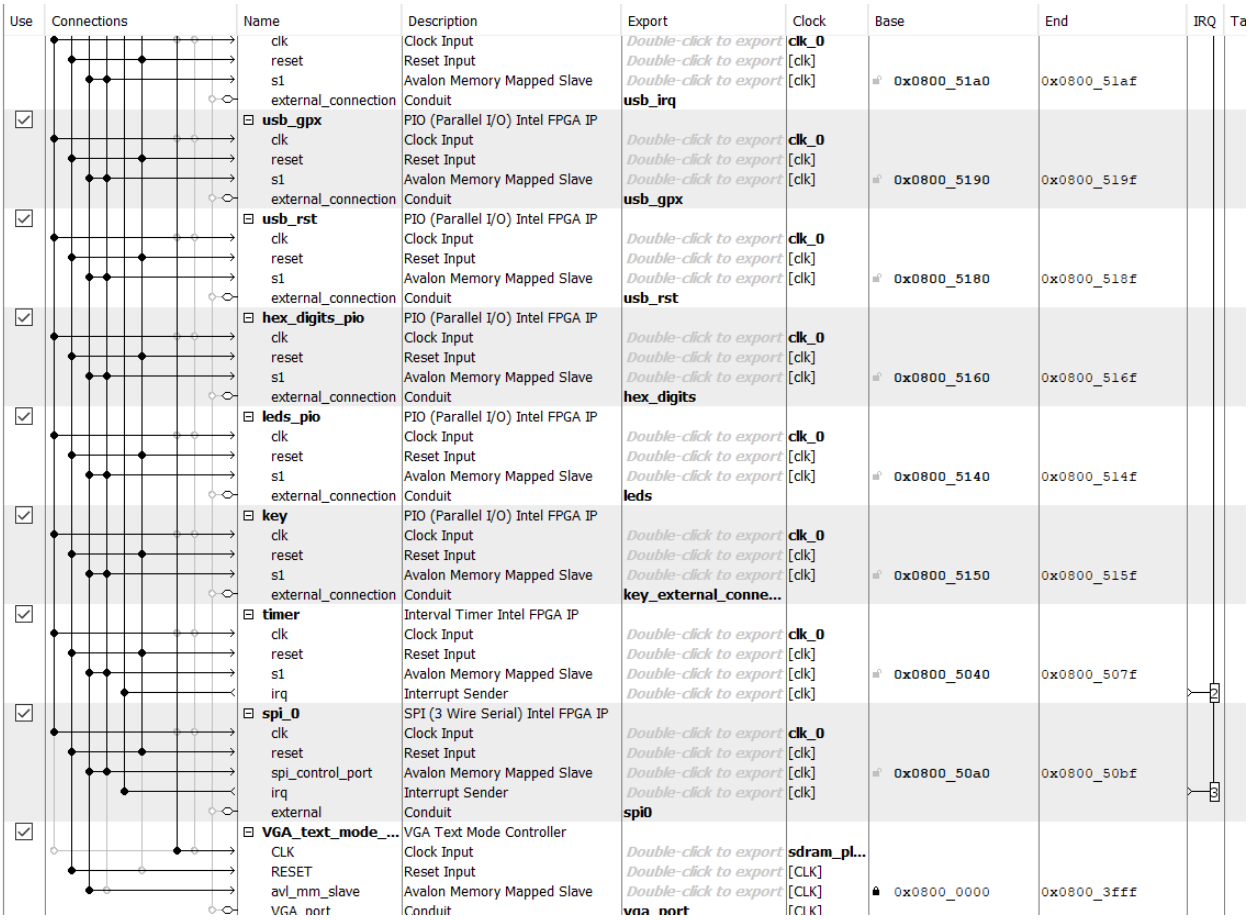


Lab 7.2 Block Diagram

Platform Designer View



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	T
☑		**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_requ...	Reset Output	*Double-click to export*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_4800	0x0800_4fff		
		custom_instructio...	Custom Instruction Master	*Double-click to export*					
☑		**sdram**	SDRAM Controller Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**sdram_pl...**				
		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_inte...				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_inte...	0x0800_51b0	0x0800_51bf		
		c0	Clock Output	*Double-click to export*	sdram_pll...				
		c1	Clock Output	**sdram_clk**	sdram_pll...				
☑		**sysid_qsys_0**	System ID Peripheral Intel FPGA...						
		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_51d0	0x0800_51d7		
☑		**jtag_uart**	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_51d8	0x0800_51df		
		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_5170	0x0800_517f		
		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_51a0	0x0800_51af		
		external_connection	Conduit	**usb_irq**					
☑		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP						



Lab 7 Platform Designer

IV. Modules and Blocks

SystemVerilog Modules

font_rom.sv

Inputs: [10:0] addr

Outputs: [7:0] data

Description: Module that holds information about each character and how each pixel is represented for each character.

Purpose: Allows retrieval of character structure information (which pixel should be colored what for each character ie. foreground or background).

HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: Module that holds information about the Hex LED displays on the FPGA board.

Purpose: Used so the correct output can be displayed onto the FPGA HEX displays.

lab7.sv

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

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

Description: Instantiates all the modules required for our design.

Purpose: Top-level for our design.

lab7_soc.qip

Description: Module that holds information about the hardware created in Platform Designer.

Purpose: Used so outside hardware can be interacted with from user.

ram.v

Inputs: [10:0] address_a, address_b, [3:0] byteena_aclock, [31:0] data_a, data_b, rden_a, rden_b, wren_a, wren_b

Outputs: [31:0] q_a, q_b

Description: Module created by Quartus GUI Megafunction to hold information about character pixel data and color data. Two-port RAM with inputs listed above.

Purpose: Used as the RAM to hold the contents of every word (replacement of the 601 registers in 7.1 except now has 1200)

VGA_controller.sv

Inputs: Clk, Reset,

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

Description: Module that sets the pixel value onto the screen using vertical and horizontal syncs.

Purpose: Used to draw or set pixel values on the 640x480 VGA screen.

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

Description: Instantiates RAM, font data, and VGA controller and gets information about foreground and background colors and which word/character is being drawn.

Purpose: Reads and writes from and to addresses in RAM or registers to determine foreground and background colors and characters being drawn.

System Blocks

clk_0

Function: Main 50 MHz clock used for the entire design and all components

nios2_gen2_0

Function: NIOS II/e processor block. It sends instructions and data to other blocks.

sdram

Function: Off-chip 512 MB memory utilizing sdram_pll as a clock to read and write operations correctly.

sdram_pll

Function: Clock delayed 1ns from main clock to accurately have sdram work.

sysid_qsys_0

Function: Ensures no mismatch or incompatibilities between hardware and software incorporation.

jtag_uart

Function: Allows software to communicate with FPGA by outputting onto terminal.

hex_digits_pio

Function: 16-bit output. Hex_digits_pio is the Hexdigit displays on the FPGA board.

keycode

Function: 8-bit input value that holds which keyboard key was pressed.

usb_irq/usb_gpx/usb_rst

Function: 1-bit PIOs that allows keyboard to communicate with FPGA.

key

Function: 2-bit input value. Key is the of the 2 buttons on the FPGA board (run and reset).

timer

Function: Clock to keep track of how much time has passed.

spi_0

Function: Allows data transfer between master and slave components.

vga_text_mode_controller_0

Function: Allows VGA signals to be linked from the Avalon bus to the VGA monitor.

V. Conclusion

Overall, we were able to get all the functionalities of the design working. Our tests were able to run properly, and our colors were correct for our palette and color test. The content for this lab can be extended to our final project for graphic drawing or just drawing to the VGA monitor in general. In lab 6, we drew our ball by a module that described everything. If we were to have multiple objects on the final project, we would need multiple modules and that can be not ideal. Having the lab 7 code will allow us to simplify the VGA output process. For the actual coding portion, there was nothing ambiguous about it. The documentation and lectures talked about the structure of memory and how the bits were ordered. The unclear portion was initializing on-chip memory. Figuring out how many ports and which boxes to check and uncheck was very time consuming. In the end, we had a graphical glitch where we couldn't find out the issue. In the end, it was because we forgot to uncheck one box when creating the RAM megafunction.

In terms of bugs, we had a fair amount. For 7.1, we had all our letters inverted when we ran the test. This was because we indexed the pixels wrong when printing to the screen and just had to reverse it to fix it. Another issue we had was that our background color was black and wouldn't change to the right color. This was fixed by looking at the FAQ and adding a check for the blank signal. This part also took a long time because we forgot to regenerate the HDL file, so

we were stuck trying to figure out a problem in the code that wasn't there. For 7.2, we had wrong colors being output on the screen. Upon further inspection, we noticed the colors were just swapped with another color. This is because in our setPalette and our color index checker in our SystemVerilog code, we flipped our values, so they were being assigned to the other color. After this fix, we had an issue where the first few pixels for every word was being put in the wrong spot. This took us a while to debug, but the issue was that we created the RAM incorrectly by forgetting to uncheck a box. Besides those bugs, no major bugs appeared during our coding of the lab.

Post-Lab Questions

	Off-Chip RAM (7.1)	On-Chip RAM (7.2)
LUT	37,588	5,606
DSP	0	0
Memory (BRAM)	46,080	119,808
Flip-Flip	22,037	3,065
Frequency (MHz)	59.61	77.61
Static Power (mW)	97.42	96.57
Dynamic Power (mW)	250.25	72.68
Total Power (mW)	369.7	191.31

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?

The on-chip RAM design is more efficient as it took significantly less time to compile. The off-chip RAM for us took approximately 16 minutes to compile while the on-chip compiled in 2 minutes. Also accessing wise, the on-chip is more efficient because it can pull data straight from an address in memory, meaning no need for additional logical units. Off-chip RAM uses registers and multiple of them must be created which takes a lot of time and power (hence the long compile time). For tradeoffs, while the on-chip is more efficient, it only has two inputs and outputs whereas the registers (off-chip) can access registers in parallel due to it having multiple inputs and outputs.