ECE 385

Spring 2023

Experiment 7

VGA Text Mode Graphics with Avalon-MM Interface

Kevin Huang (kuanwei2), Steven Chang (sychang5)
TY 10:15 A.M.
Tianhao Yu

Introduction

In Lab 7, we built the project on top of the design of Lab 6.2 with the Color Mapper and VGA controller modules used. In Lab 7.1, we utilized Avalon Memory Mapper bus to read and write 32-bit data of VRAM into registers depending on the NIOS-II processor, which is the master of Avalon MM. The 32-bit data in VRAM is encoded in little-endian form, where the least significant bits contain the character that will be displayed first, and each 32-bit data is separated into 8-bit for each character, where the most significant bit indicates whether the character is inverted or not and the last 7 bits are the glyph code for the character. To display the entire VGA monitor, we need 600 registers to store the information for all 2400 characters, as there are 80 columns x 30 rows of words, and each word is in the size of 8 x-pixels x 16 y-pixels. More importantly, an additional register called the control register is used to indicate the foreground and background RGB values. For Lab 7.2, instead of storing the data in registers, we relocated the data into on-chip memory as we have 16-bit data for each character, which is doubled the original size and is no longer fitting in the FPGA registers. Thus, each 32-bit VRAM only stores the information of 2 characters, where the most significant bit for each 16 bits is the indicator for the invert, followed by 7 bits of glyph code, 4 bits of foreground index, and 4 bits of background index. On the other hand, instead of having a control register, we created a Color Palette with 8 32-bit registers to store RGB values, where each register stores the RGB for two colors. Our lab 7 designs were an extension of the week 2 lab 6 designs in the sense that the we built off of the concept of using the VGA interface but instead of simply drawing a ball on the screen, we wrote text to the screen with its corresponding colors. In other words, we essentially increased the number of parameters to include character information and color scheme in order to correctly draw on our monitor.

Written Description of Lab 7 System

Week 1 (Monochrome Text Display)

I. Written Description of Lab 7 System

The hardware aspects of our design for lab 7 are similar to those for lab 6, which include the NIOS-II processor, SDRAM controller, SDRAM PLL, on-chip memory block, as well as a number of PIO (parallel input/output) peripherals, the JTAG UART peripheral, and an SPI protocol. More importantly, for lab 7.1, we utilize Avalon Memory Mapper to allow the NIOS-II processor, which is the core component that communicates with the peripherals through the Avalon Bus, to read and write data to the registers. The color codes for each pixel on the VGA monitor are then stored in the registers. Since lab 7 doesn't use USB port to access devices, such as the keyboard, the System ID checker is no longer needed. The JTAG UART peripheral enables us to use the terminal of our computer running the program to communicate with the NIOS-II and crucially print statements that help with debugging.The SDRAM serves as the storage component for the code and data of our program, while the on-chip memory block is unused for week 1 of the lab. The SDRAM PLL helps compensate for the clock skew and ensures that the SDRAM runs properly with the SDRAM clock connected to the output of the SDRAM PLL. For week 1, the glyph code, foreground, and background color are stored in the registers to draw out the specific words with different colors. For the week 2 program, we moved

all the data from registers to on-chip memory as each word now has 16-bit of data, including 1-bit of an inverter, 7-bit of glyph code, 4-bit of foreground index, and 4-bit of background index. However, we kept the color palette registers local with a register-based approach, with 8 registers holding 16 color schemes in total.

II. VGA Text Mode Controller IP Description

In lab 7, the VGA text mode controller IP interfaces with the NIOS II through the Avalon-MM and outputs corresponding signals including the VGA port signals to correctly output to the VGA monitor. It includes the Avalon-MM slave port as mentioned that will complete read and write operations as instructed by the NIOS-II processor (master). These read and write operations handle the character glyph code information as well as the color schemes for the display. In week 1, we utilized a register-based approach, then shifted to an on-chip memory approach in week 2. Information stored/accessed through these read and write operations include the foreground and background colors of various color schemes, glyph code of characters, as well as inverse bits to signal whether foreground or background colors should be used. In addition to the registers/memory storing character information, within the VGA text mode controller, we also include instantiations of the font ROM (includes character data, accessed based on character glyph code and electron gun coordinates) and VGA controller (which provide electron gun coordinates, essentially signaling where we are drawing on the monitor) in order to correctly write characters to the display through additional logic.

III. VGA Register Read and Write

In week 1, we utilized a register-based VRAM approach. Therefore, we declared 601 local registers with 32 bits each as follows.

```
logic [31:0] LOCAL_REG [601]
```

The 0th to 600th registers (inclusive) are used to store character information, including character glyph code and its inverse bit. The 601st register is the control register that stores the foreground and background colors. The read and write operations to the registers are encompassed within an always_ff block in order to account for the read waitstate of 1 to ensure correct data is retrieved. The control signals include AVL_READ, AVL_WRITE, AVL_CS, and AVL_BYTE_EN. Reading and writing from and to the registers should only be performed when AVL_READ and AVL_CS or AVL_WRITE and AVL_CS are both high, respectively. If we are performing a read operation, then we can simply set AVL_READDATA to be the content of the local register corresponding to AVL_ADDR. If we are performing the write operation, then we will need to check the AVL_BYTE_EN values to determine whether we are writing all 32 bits of data into the local register (once again determined by AVL_ADDR), or specific bits only (like the first 8 bits, last 8 bits, etc.). A byteenable description table is shown below for reference. To support read OR write operations, not both at the same time, we placed condition checks using else if statements instead of if statements. In addition, we also included a reset option on a high RESET signal to the text mode controller module, which would go through each of the 601 registers and clear the data to all 0s.

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.

Figure I. Byteenable Description/Action

IV. Draw Text Characters from VRAM and Font ROM Algorithm

In order to draw the correct text characters on the VGA monitor, we needed to correctly compute the address with which we access VRAM and font ROM at any point on the screen. The VRAM contains information about the character glyph code as well as whether or not the color of the character should be inverted, while the font ROM sprite table contains the data that determines whether we should assign the foreground or background color for each pixel. To correctly access VRAM and font ROM, we need to compute their corresponding addresses with which we access. To do so, we first compute the column in which the character resides by dividing DrawX by 8 (or right shift 3 times), as well as the row in which the character resides by dividing DrawY by 16 (or right shift 4 times). This is because each character is 8x16 pixels, with a total of 80 columns and 30 rows on the screen. The address into the VRAM is then computed by multiplying row by 80 (column size) then added with the column, all of which is then divided by 4 (or right shift by 2). **VRAM Address = (row*80 + column)/4** Essentially, we are accessing using row major order. The division by 4 is necessary because 4 character information (character glyph code and inverse bit) are stored within each register. Next, the font ROM address will be determined by DrawY and the data retrieved from the register VRAM, but the specific bits that will be used from the data from the VRAM will be dependent on which shape/character we are drawing. However, the general equation is font **ROM Address= DrawY - shape_y + 16*(7 bits of data)**, where shape_y is the row left shifted 4 times (times 16). The 7 bits of data is dependent on which character we are drawing. For instance, if we are drawing the first character in the group of 4 characters, then we will take the first 7 bits of the data from the VRAM, if we are drawing the second character in the group of 4 characters, then we will using the bits 8 through 14 of the data from VRAM, and so on. The DrawY - shape_y essentially guarantees that we correctly access the right row in the font ROM of each character that we draw, and the multiplication by 16 is because each character/symbol takes up 16 rows of information in the font ROM sprite table.

With an understanding of how we retrieve data from our VRAM and font ROM, we modified the lab manual approach to drawing text on the VGA monitor. After finding the necessary information like column number, row number, VRAM address and VRAM data inside an always_comb block, we used another always_comb block to determine which shape/character

out of the group of 4 characters we were drawing at any point on the screen. We determined this by looking at the least 2 significant bits of the column number. If col[1:0] was 00, it meant we were drawing character 1 (leftmost character), if it was 01, it meant we were drawing character 2 (second leftmost character), and so on. Within these checks, we set the corresponding flags that will help us determine which character we were drawing. It also helped us determine the correct font ROM address based on which character we were drawing, using the formula discussed above. Onto the actual 'drawing portion', we first determined which shape/character we were drawing by checking the flags we set previously. Then, nested within that check, we looked at the font ROM data retrieved in reverse order (because we are drawing from left to right) to see if each bit is 1, if it was, then we set RGB as the corresponding foreground RBGs if the inverse bit is 0, and RGB as the corresponding background RGBs if the inverse bit is 1. The inverse bit is dependent on which shape/character we were drawing. If we were drawing the first character in the group of 4, then its inverse bit is the 7th bit of the VRAM data, opposed to the 15th bit of VRAM data if we were drawing the second character, and so on.

V. Implementation of the Inverse Color Bit & Control Register

As mentioned previously, the control register is the 600th local register (0 indexed) that we declared in a register-based VRAM approach. Looking at the bit-encoding of the control register, we retrieved the RGB of the foreground and background colors in an always_comb block for ease of reference in the drawing portion. Referencing the bit-encoding, we set the BKG_B (background B) to be bits 1 through 4 (0 indexed) of the 600th local register, BKG_G (background G) to be bits 5 through 8 of the same register (control register), and so on, with respect to the bit encoding provided. Within the actual drawing portion of code, we first checked if we were drawing anything (if the font ROM data is a 1 bit for that specific pixel), and if we were, checked if the inverse bit was high (signaling we should inverse the colors), which is character dependent and found in the VRAM data as discussed above. If the inverse bit was low, then we set RGB to the foreground RGB, if the inverse bit was high, then we set RGB to the background RGB. The opposite is true if the font ROM data is low for that specific pixel. In other words, we had 4 possible cases of RGB colors for each character, depending on the factors of if font ROM bit is high or low and if the inverse bit is high or low.

Week 2 (Color Text Display): Hardware Changes to Support Multi-Color Text

I. Modification of Register-Based VRAM to On-Chip Memory-Based VRAM

As week 2 lab has to be able to support multi-color text, the color information for each character has now changed from 1 byte to 2 bytes, which include 1-bit of an inverter, 7-bit of glyph code, 4-bit of foreground index, and 4-bit of background index. Therefore, instead of 600 registers, week 2's lab requires 1200 registers just for the character information alone. If we maintained the same register-based VRAM approach for week 2, we would not have sufficient resources. Therefore, we shifted to an on-chip memory based approach to VRAM for week 2. This drastically reduced the cost of our design and made the overall program more efficient at implementation. In addition, instead of having one control register contain the color information, we now extended it to 8 32-bit registers to support a 16-color palette, which is accessed based on the foreground and background index of the character information stored in the VRAM. One

important aspect of on-chip memory to note is that while registers have as many input and output ports as storage bits, memory has fixed and limited number of input and output ports. In our specific case, we only have 2 ports, to support two read or write operations at any point in time simultaneously. For our design, one of the ports was used to support the Avalon-MM read and write operations, while the other port was used specifically for read operations to retrieve character data information (including glyph code and color indices). Due to this limitation, we kept a register-based approach to our color-palette and used 8 32-bit registers to store the 16 color schemes required for the design.

II. Platform Designer Modifications

In term of modifications to the platform designer, not much was needed to transition from week 1 to week 2. The only major change that we made was change the width of the AVL_ADDR of the text mode controller from 10 bits to 12 bits in order to account for the increase in VRAM to support the multi-color palette. The difference between when to access the color data and the VRAM will be determined by the most significant bit of the AVL_ADDR, with a high AVL_ADDR 11th bit signaling that we should access the local color registers and a low bit signaling that we should access the VRAM. This is because the color palette begins at the word address of 0x800, which has 11th bit of 1. Lower addresses include unused but reserved locations as well as the VRAM data containing our character information.

III. Modified Sprite Drawing Algorithm

Similar to week 1, we compute the current col with DrawX/8 and the current row with DrawY/16, and then the current order for the character is computed by row*80 + col as there are 80 characters in each row. However, different from week 1, instead of dividing the current order by 4 to get the address of the current register, we divide it by 2 instead because in week 2, each register only stores the color information for 2 characters instead of 4. Since each character now has 16-bit of color information, each register now stores 1-bit of an inverter, 7-bit of glyph code, 4-bit of foreground index, and 4-bit of background index for each of the two characters. Therefore, the updated indexing equations/**VRAM Address = (row*80 + column)/2**. The font ROM address has a similar general formula to week 1, with **font ROM Address = (DrawY - shape_y + 16*(7 bits of data)**. The 7 bits of data now corresponds to different bits of the data accessed from VRAM. If we were drawing character 1, then we would access the 8 through 14 bits of the VRAM data, but if we were drawing character 2, then we would access the 24 through 30 bits of the VRAM data. To decide which of the two characters to draw in the current register, we'll use the least significant bit of the current order computed before to determine where a value of 1 indicates to use of the color information stored in the most significant 16-bit while a value of 0 indicates to draw the character stores in the least significant 16-bit based on the little-endian form. More importantly, as week 2 stores 16 colors in a color palette formed by 8 32-bit registers, each register now stores color information for two colors with 4 bits of R, 4 bits of B, and 4 bits of G for each color. In week 2, the algorithm is modified to use the foreground and background indexes in each register to determine which color register to use. Since each color register stores two colors, we also divide the index by 2 to find out the address for the color register. Then, similarly, we use the last bit to determine which of the two colors in the color register to use.

IV. Additional Modifications

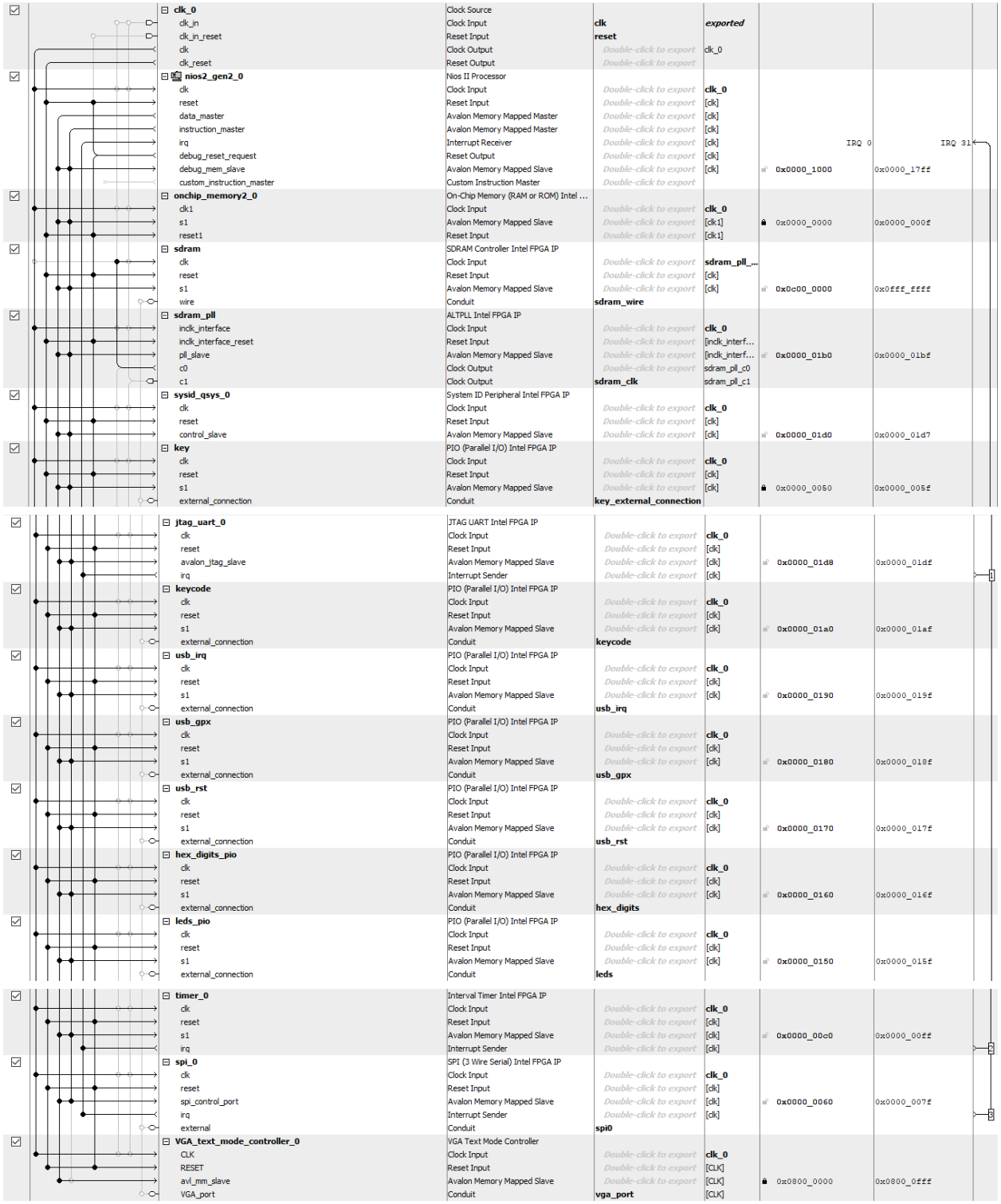
In terms of additional modifications necessary, the main modification comes down to how we access the colors for our characters. As mentioned previously, the limitations of the on-chip memory based VRAM resulted in us storing the color palette registers locally, with 8 32-bit registers storing 16 color schemes. In order to correctly access the right colors for each character and truly support multi-color character display, the character information contained in VRAM now changed to include the foreground and background index for each individual character, with each index corresponding to 0-15 of one of the color schemes stored in the color registers. This modification manifested in the form of modified code, which will be discussed below.

V. Additional Hardware/Code to Draw Paletted Colors

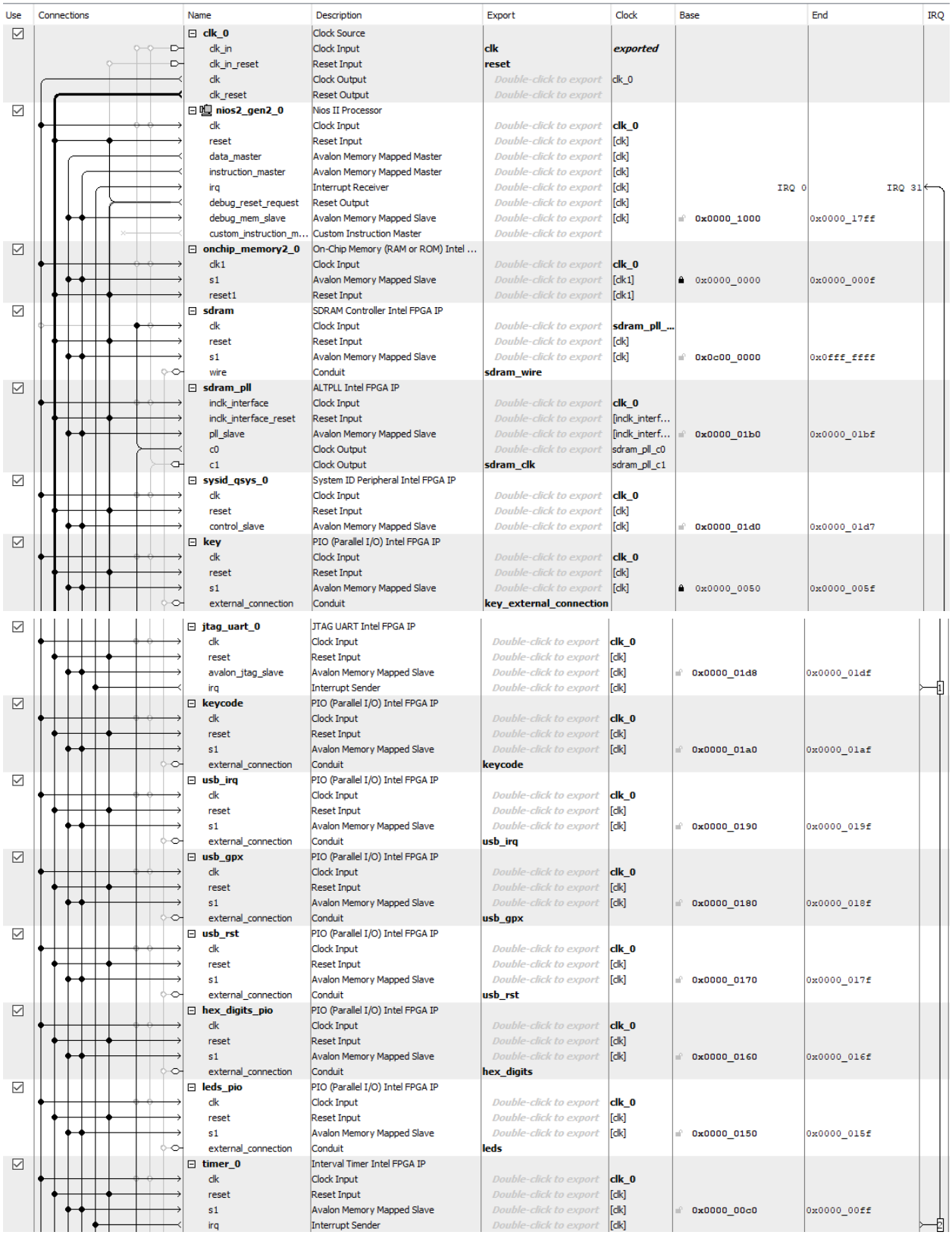
In order to support multi-color text display, we first needed to create the color palette register write operation. This is fairly straightforward and a direct modification of the week 1 code supporting reading and writing to the 601 local registers. We would write to the color palette registers if AVL_CS, AVL_WRITE, and AVL_ADDR[11] were all high simultaneously, storing the AVL_WRITEDATA into the color register given by the least significant 3 bits of the AVL_ADDR. Note that we use the most significant bit of the AVL_ADDR to determine whether we are accessing the color registers or VRAM, which was discussed above. In addition, another necessary modification was the assignment of RGB colors for each character. We made the modification of setting RGB for each character at each pixel by first accessing the correct color register (0 through 7) with the data retrieved from the VRAM, then we picked whether to use the upper or lower color stored in each color register based on the least significant bit of the color index of each character.

VI. System Level Block Diagram

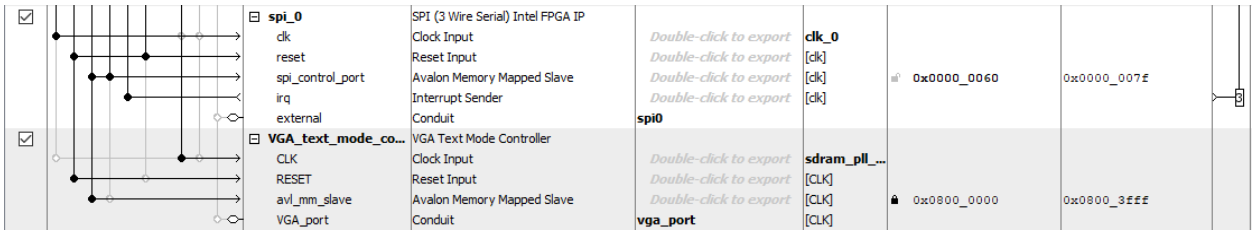
Lab 7.1 Platform Designer



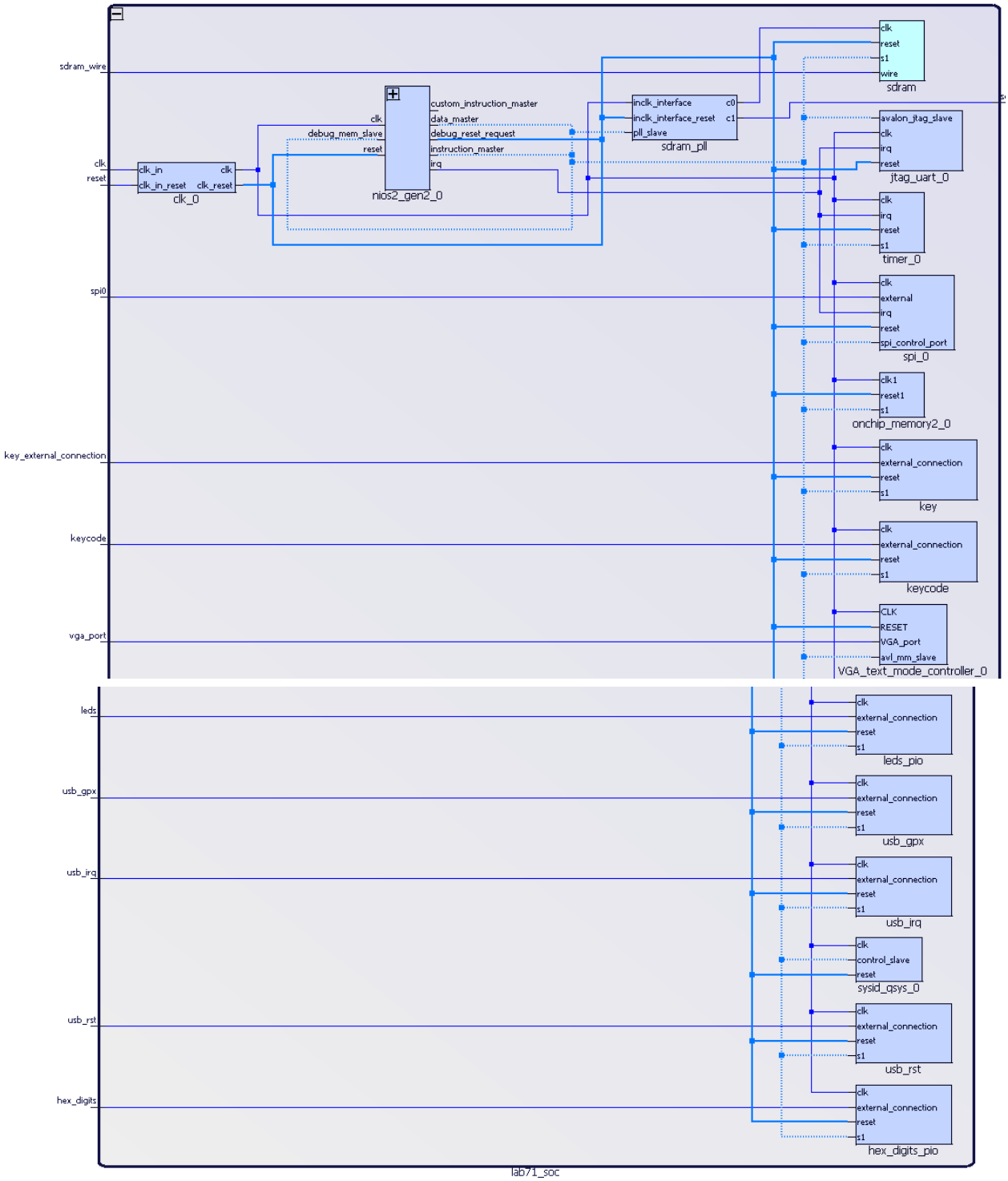
Lab 7.2 Platform Designer



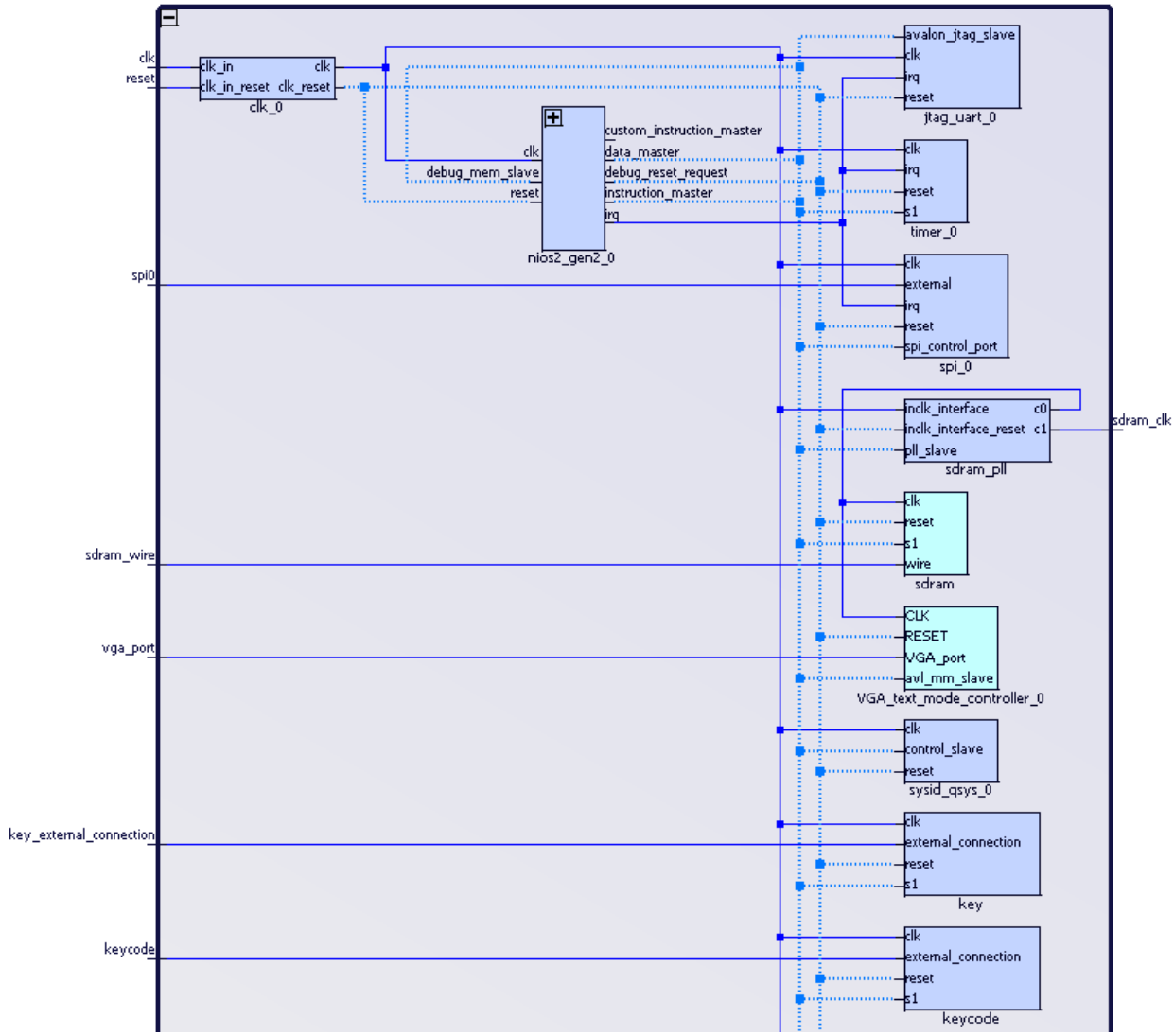
Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☑		**clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	*Double-click to export*	clk_0			
		clk_reset	Reset Output	*Double-click to export*				
☑		**nios2_gen2_0**	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
		debug_reset_request	Reset Output	*Double-click to export*	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	0x0000_17ff	
		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]	0x0000_0000	0x0000_000f	
		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]	0x0c00_0000	0x0fff_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...	0x0000_01b0	0x0000_01bf	
		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]	0x0000_01d0	0x0000_01d7	
☑		**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]	0x0000_0050	0x0000_005f	
		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 Slave	*Double-click to export*	[clk]	0x0000_01d8	0x0000_01df	
		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]	0x0000_01a0	0x0000_01af	
		external_connection	Conduit	**keycode**				
☑		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0190	0x0000_019f	
		external_connection	Conduit	**usb_irq**				
☑		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0180	0x0000_018f	
		external_connection	Conduit	**usb_gpx**				
☑		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0170	0x0000_017f	
		external_connection	Conduit	**usb_rst**				
☑		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0160	0x0000_016f	
		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]	0x0000_0150	0x0000_015f	
		external_connection	Conduit	**leds**				
☑		**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]	0x0000_00c0	0x0000_00ff	
		irq	Interrupt Sender	*Double-click to export*	[clk]			2

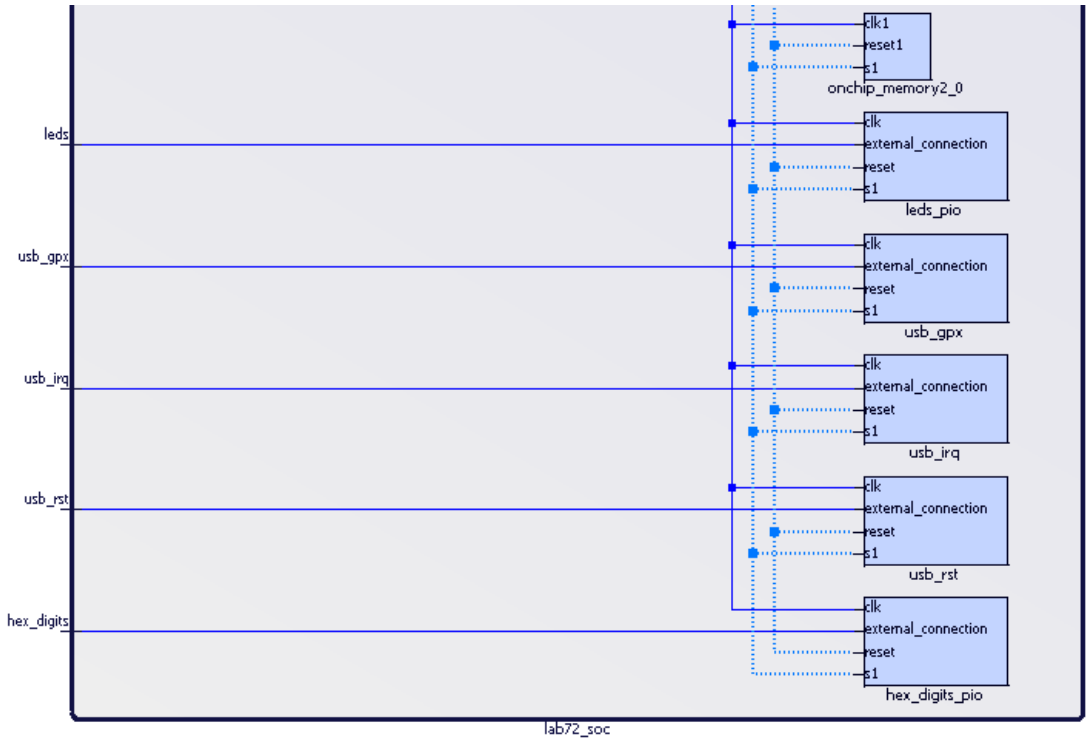

spi_0
SPI (3 Wire Serial) Intel FPGA IP
clk
Clock Input
Double-click to export
clk_0
reset
Reset Input
Double-click to export
[clk]
spi_control_port
Avalon Memory Mapped Slave
Double-click to export
[clk]
0x0000_0060
0x0000_007f
irq
Interrupt Sender
Double-click to export
[clk]
external
Conduit
spi0
VGA_text_mode_co...
VGA Text Mode Controller
CLK
Clock Input
Double-click to export
sdram_pll_...
RESET
Reset Input
Double-click to export
[CLK]
avl_mm_slave
Avalon Memory Mapped Slave
Double-click to export
[CLK]
0x0800_0000
0x0800_3fff
VGA_port
Conduit
vga_port
[CLK]

Lab 7.1 System Level Block Diagram



Lab 7.2 System Level Block Diagram





clk_0 - System clock with 50 MHz from the FPGA that is used to clock and reset other modules.

nios2_gen2_0 - 32-bit CPU that allows the developers to code and debug in C for the processor and send out designated control signals to modules.

sdram - the sdram is the off-chip memory block that is used to store code and data. It is interfaced with the NIOS-II processor through the memory controller that helps manage data transfers. It effectively serves as the bridge between data transferred between the processor and the SDRAM.

sdram_pll - Provides clock signal to compensate for the propagation delay that may arise between the input and output clocks, making them synchronized.

sysid_qsys_0 - This is used to ensure the hardware and software are compatible and updated to avoid any error in the connections between them.

jtag_uart__0 - The JTAG UART provides a communication interface between the FPGA and other devices. In this lab, it allows the user to use “printf” on the console for debugging.

Lab 7.1:

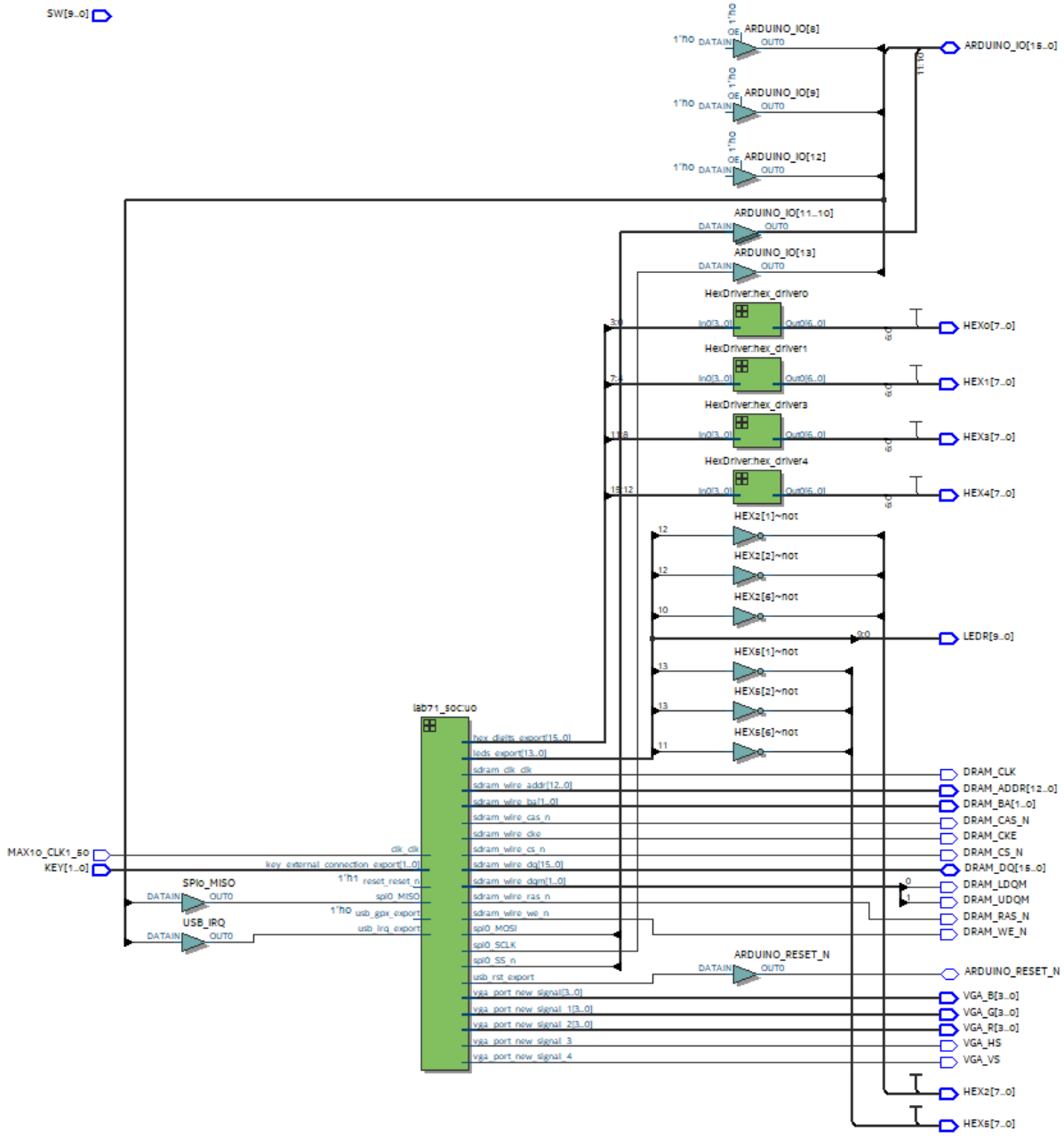
VGA_text_mode_controller - This provides the Avalon Memory Mapper for the NIOS II to access the content of the VRAM and to correctly output necessary data to output to the VGA monitor.

Lab 7.2:

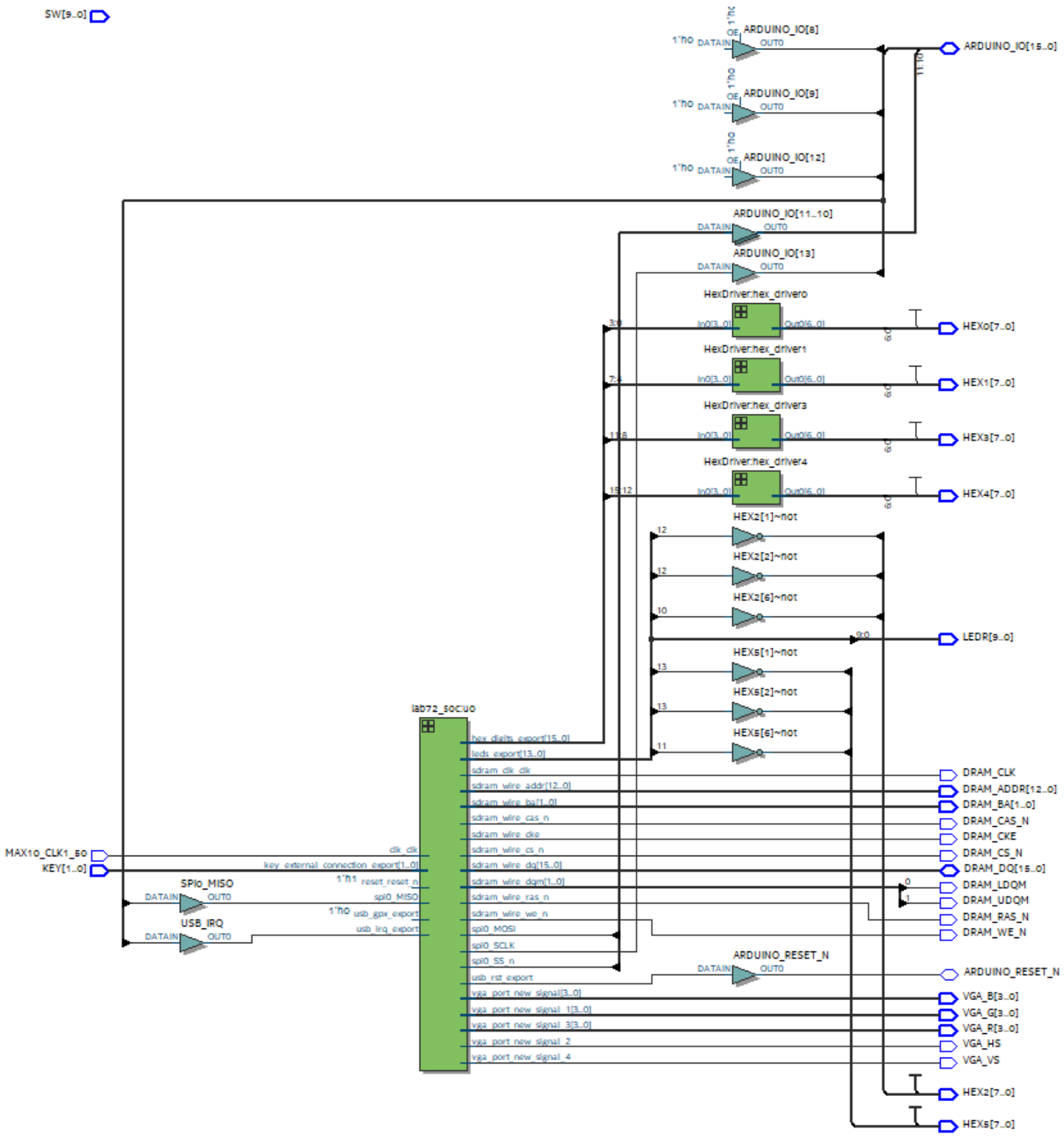
Onchip_memory2_0 - On-chip memory uses to character information including the foreground and background color indices as well as the character glyph code and inverse bit.

Top Level Block Diagram

Week 1 Top Level Block Diagram



Week 2 Top Level Diagram



SV Modules Summary

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, [31:0] AVL_READDATA
Outputs: hs, vs, [3:0} red, [3:0] green, [3:0] blue
Description: This module declares on-chip memory and submodules, such as VGA Controller and ROMS. More importantly, it is used to compute the current drawing row and column for font rom module to determine the current glyph. Then, the AVL_WRITE will help to determine when to write data into the color registers, where the RGB values for different colors are stored. Last but not least, based on the current glyph, DrawX, and DrawY, the module will extract the foreground and background colors from the color register according to the code it gets from the data and saves the RGB value as the output.
Purpose: This module is used to determine the RGB values of the current drawing pixel based on the current position and the code inside the data that is stored in on-chip memory.

Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY
Description: This module contains a positive-edge triggered flip flop that will set hc and vc, which are the current drawing pixels to 0 whenever Reset is high. Otherwise, it will then increase hc by 1. If hc reaches the max for horizontal pixel, then it will be set to 0, and vc is incremented by 1. It also sets pixel_clk to the current clkdiv. Not to mention, the module will modify the hs and vs, which are the sync pulses for the display, depending on the current drawing pixel. Last, the display signal will be high only when the hc is between 0-639 and vc is between 0-479.
Purpose: This module determines the sync pulses and pixel_clk for VGA display. It will also run through each pixel and calculate the DrawX and DrawY and blank signals for the display.

Module: font_rom.sv
Inputs: [10:0] addr
Outputs: [7:0] data
Description: This is the font rom module that stores the data for 16x8 pixels of each 128 glyphs. The module will return the corresponding 8-bit data depending on the 11-bit input addr.
Purpose: This module is used to determine the pixel value of the current drawing glyph.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the Hex driver that takes in 4-bit value In0 from the register to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.
Purpose: This module is used to show the value in the adder in the LED on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

Module: lab7.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW

Outputs: [7:0] LEDR, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, DRAM_LDQM, DRAM_UDQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B
Inouts: [15:0] DRAM_DQ, [15:0] ARDUINO_IO, ARDUINO_RESET_N
Description: This module instantiates module lab7soc, which was generated by Platform Designer, and all the connections on FPGA, such as the clock, I/O, and SDRAM signals.
Purpose: This module is used as a top-level to connect the Nios II system to the rest of the hardware.

Post-Lab Questions

1. Fill in the table shown in IQT.30-32 with your design's resource and statistics.

Lab 7.1 Design's Resource and Statistics

LUTs	32625
DSP	0
Memory (BRAM)	55,296 / 1,677,312 (3 %)
Flip-Flop	22,049
Max Frequency	66.84 MHz
Static Power	96.18 mW
Dynamic Power	247.68 mW
Total Power	324.17 mW

Lab 7.2 Design's Resource and Statistics

LUTs	4541
DSP	0
Memory (BRAM)	202,752 / 1,677,312 (12 %)
Flip-Flop	2949
Max Frequency	77.63 MHz
Static Power	96.18 mW
Dynamic Power	76.75 mW
Total Power	189.22 mW

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

One immediate difference between the design resources and statistics table that we notice is the dramatic decrease in number of LUTs and flip-flops from week 1 to week 2. However, we should also note the corresponding increase in memory/BRAM from 3% to 12%. This is directly related to our shift in design approach from a register-based VRAM to an on-chip memory based

VRAM. With this change in approach, we also witnessed an increase in maximum frequency, as well as a decrease in total power, which is as expected. In other words, the on-chip memory approach to the design is more efficient and consumes less power. The tradeoff is the increase in memory usage.

Document any problems you encountered and your solutions to them.

One of the major problems that we encountered was during week 2 of the lab. After we correctly displayed the text, one issue that we noticed was that we were displaying exactly half of the colors, with the other half missing/not properly displaying when they were supposed to. Fortunately, this was an easy fix as we remembered that we did not increase the read wait state of our VGA text mode controller to 2, which meant there wasn't sufficient time to correctly access the other half of the pallette as each palette contained information of two color schemes.

Conclusions

Through this lab, we learned about how to create a simplified VGA text mode controller IP core to draw up to 80 column and 30 rows of text onto the VGA monitor. In week 1, we designed the text mode controller with a register-based approach to VRAM and supported only monochrome text display. In week 2, we learned how to transition from a register-based VRAM approach to an on-chip memory based VRAM approach, and further extended the monochrome text display to support color text utilizing color palette registers to support up to 16 different color schemes.

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

As mentioned above, the only major issue that we encountered was during week 2, when we were not correctly displaying all possible color schemes for our text when necessary. To fix this issue, we simply increased the wait state of our read operation to 2 in the VGA text mode controller in order to have sufficient time to access both color schemes stored within each register.

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

The character display on the VGA monitor could be extended for our final project to help display descriptions of special elements in our game. The usage of the font ROM sprite table could also be extended to help draw specific graphics, as well as the start menu and end game screens.

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 does not get changed.

No, we did not find anything particularly ambiguous or incorrect in the lab manual or given material.