ECE385 Experiment 7: Lab Report

Spring 2023
Eshaan Tibrewala (eshaant2) and Shivam Patel (shivamp6)
TA: Shitao Liu

Introduction

The objective of Lab 7 was to design a simplified text mode graphic controller that utilizes the Avalon memory mapped bus and supports 80 column text mode through the VGA output. In the first week, we created a simplified version of the graphics adapter to see if we could display text from our ROM to the monitor. For simplicity, week 1 was mainly black and white text and focused little on the actual colors. However during the second week, we extended our design so that our graphics controller utilized on-chip memory, so that we could access a much larger video memory. We also created a color palette so that we could further expand our graphics controller to support more colors.

Our overall design for Lab 7 is built on our Lab 6 design, during which we were first introduced to graphics and VGA. In Lab 6, our objective was to generate a ball on the monitor and then use keyboard inputs to control the ball on the screen. We were able to accomplish this creating the actual ball and color mapping logic in System Verilog, and then using C code to control keyboard inputs into the FPGA by using the USB shield on the FPGA. Similar to Lab 6, we used the same `VGA_controller` module to generate the 25MHz pixel clock, and instantiate the horizontal and vertical counters that are used to draw the actual pixels onto the screen or monitor. We also used a similar drawing logic in Lab 7 as we did in Lab 6, however we had to expand our coloring logic for this lab because

we utilized a color palette instead of hardcoding specific colors. This required us to build on our Lab 6 because we had to design a way to store the palette, and then read the data and extract the actual foreground and background colors from the memory location where the palette data was stored.

Written Description of Lab 7 System (shivam)

Week 1 Monochrome Text Display

Written Description

During the first week of the lab, we designed SystemVerilog code that would display an 80-column text mode with alternating colors on the VGA monitor on an infinite loop. We created a simplified version of the graphics adapter to see if we could display text from our ROM to the monitor. In the second week, we designed a screensaver that would print a sentence in a random place on the monitor with the foreground and background colors taken from a 16-color palette. We used either local registers or the FPGA on-chip memory to access data for how to color an 8x16 pixel glyph and what character that glyph should be. In week one we had data for four glyphs to be drawn, including if foreground or background should be drawn. In week two, however, we were instead given data for two glyphs but now we had data for which specific color to draw rather than just foreground or background.

VGA Text Mode Controller IP

Our VGA Text mode controller IP has four parameters: CLK, RESET, avl_mm_slave, and VGA_port. The clock and reset inputs are driven to CLK and RESET on the NIOS II processor respectively and the avl_mm_slave is used to interact with the Avalon memory-mapped bus. The VGA_port is used to send information on the necessary codes and ASCII characters back and forth between the VGA text mode and the output device so the correct characters can be drawn onto the screen.

Read/Write Logic for VGA Registers

Read and write were crucial in the proper implementation of the lab since we needed to perfectly access the VGA registers otherwise we would get the wrong data. We first put but in an always_ff block that only reads or writes if the AVL_CS signal is high. Inside that condition, we had a condition such that if read was high we would assign the value in the register to AVL_READDATA. We also had another condition such that if write was high, we would write whatever the byte enable chooses. We included a case statement based on the AVL_BYTE_EN signal that would load specific bytes into the register.

Draw Algorithm

Since our screen was 640x480 pixels and we had glyphs that were 8x16 pixels, we wanted to represent our screen glyphs. To do this, we divided 640 by 8 and by 480 by 16 to represent our entire screen as 80x30 glyphs. To do this in code, we right-shifted `DrawX` by 3 bits and stored it in `CharX` and then right-shifted `DrawY` by 4 bits and stored it in `CharY`. We then had to access which register to access and which character in that register to access but we first had to create a general representation of the screen. We used the row-major order formula and stored $80 * charY + charX$ in `rmo`. We could now find our register index and our character index. We did `rmo` divided by 4 for `regidx` and `rmo` mod 4 to find `charidx`. After we did this we now had which register of the 600 we wanted to access and which one of the 4 characters we wanted to access. After getting the 32-bit value from `local_reg[regidx]` we sliced it based on what our `charidx` was, leaving us with an 8-bit character code. From here, we appended the bottom 7 bits of the character code with our `DrawY` mod by 16 and passed that appended variable into our `fontram` as the input. The 7 bits of the character code told `fontram` which one of 128 characters we want to draw as our glyph and $DrawY \ \% \ 16$ picks one of the 16 rows to output and store in `fontdata`. From here, we then proceeded to choose the color and draw it onto the VGA monitor.

Inverse Color Bit Implementation

In our code, the output of our fontrom was put in fontdata, which represents one of the sixteen rows of 1s and 0s that creates our glyph. The most significant bit of the character code in our character code from the local_register is our invert but so we included that as a condition in our draw logic. In our draw logic, we initially set red, green, and blue to 0 and then had an if statement for blank. We only want to draw if blank is 1 otherwise we do

nothing. Nested in that condition was another for the inverse bit. In fontdata, the 1s mean that we draw the foreground color and the 0s mean that we draw the background color and if the invert bit is high we want to do the opposite. We included this logic by saying if the invert bit is high we draw the foreground color with 0 and the background color with 1 otherwise we assign color normally. Finally, we accessed the 601st register, which is the control register, and assigned red, green, and blue to the foreground RGB or background RGB values by slicing the bits of the control register accordingly.

Week 2 Color Text Display

Hardware Changes for Multi-Color Text

Registered Based VRAM Modifications

In the first week of the lab, we used an array of 601 registers but for the second week, we had to change to using the system-on-chip memory. We changed it such that we can store the character code and also the pixel data which we needed to draw on the screen. It used two ports to read and write data into the SOC. Since we now had to omit the implementation of the local registers, we changed the VGA controller implementation such that we directly read the character codes from the on-chip memory.

Corresponding Modifications to Platform Designer IP

Since we had to change our use of local registers to on-chip memory, we had to first instantiate it and configure its ports and connect any paths necessary. We changed the addresses being accessed in the C code so it could access the correct corresponding RAM values. We also connected it to the IP blocks through the platform designer to make sure that all of the connections were accurate so we could proceed with developing our code.

Modified Sprite Drawing Algorithm

We had to make small changes since each of our registers had only two glyphs now instead of four. In week 1, we divided our row-major order variable (rmo) by four to choose our register and modded it by four to choose the character and then store that corresponding 8-bit character code in a variable. We changed this to dividing rmo by two to choose our register and modding it by two to choose the character instead of storing a 16-bit glyph code in a variable. From here we had to separate the glyph code from the color code so we created 4-bit foreground and background variables that would serve as

indexes to figure out which of the 16 colors in our palette we want to draw. We bit-sliced to store the character code in drawVal to be passed into fontrom and we also bit-sliced to store our foreground and background values. Since we needed to access our color palette we created 8 local registers each to store two colors and then had to change our read and write logic. We changed the AVL_WRITE condition such that the value of AVL_WRITEDATA would be loaded into the COLOR_PALETTE when AVL_WRITE is high. We also changed the AVL_READ condition such that the COLOR_PALETTE would be loaded into AVL_READDATA if the AVL_READ signal was high. The other logic from week one remained the same except for the drawing logic. Since we did not have the control register anymore we had to now draw from the color palette and this required new logic. Based on the output of fontrom we had a condition to see if the pixel had a foreground or background color. If it was foreground, we set the variable “temp” equal to the foreground variable value we had set earlier otherwise we would set temp equal to the background variable. To now find out which of the sixteen colors in the 8 local color palette registers to use, we can use temp as an index. Dividing temp by 2 and passing that through would give us which register to choose and mod temp by 2 would give us which of the two colors in the register to choose. We stored COLOR_PALETTE[temp/2] in our colorVal variable and then we sliced it accordingly to assign our RGB values.

Block Diagram

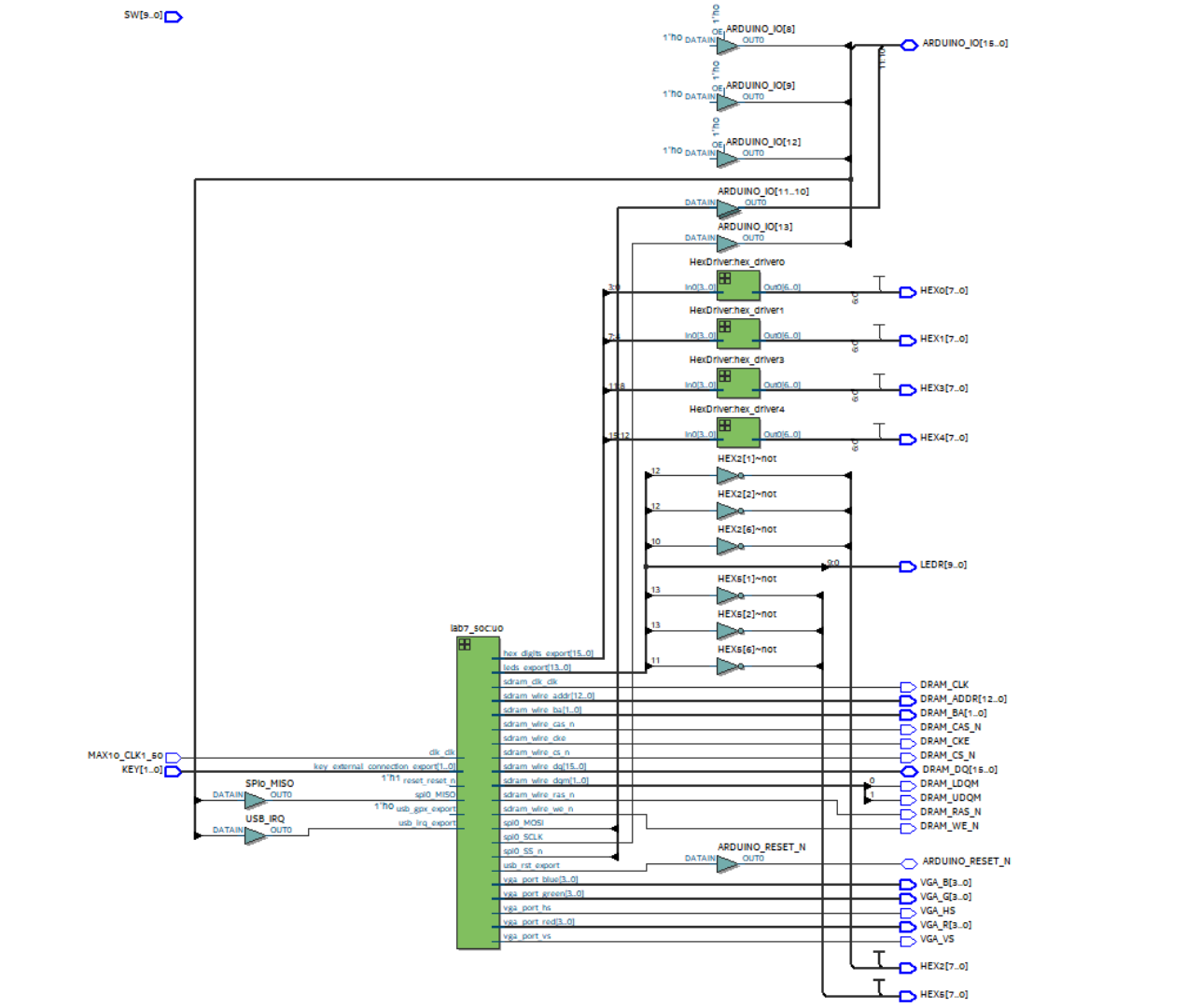


Figure 1: Week 2 Top Level

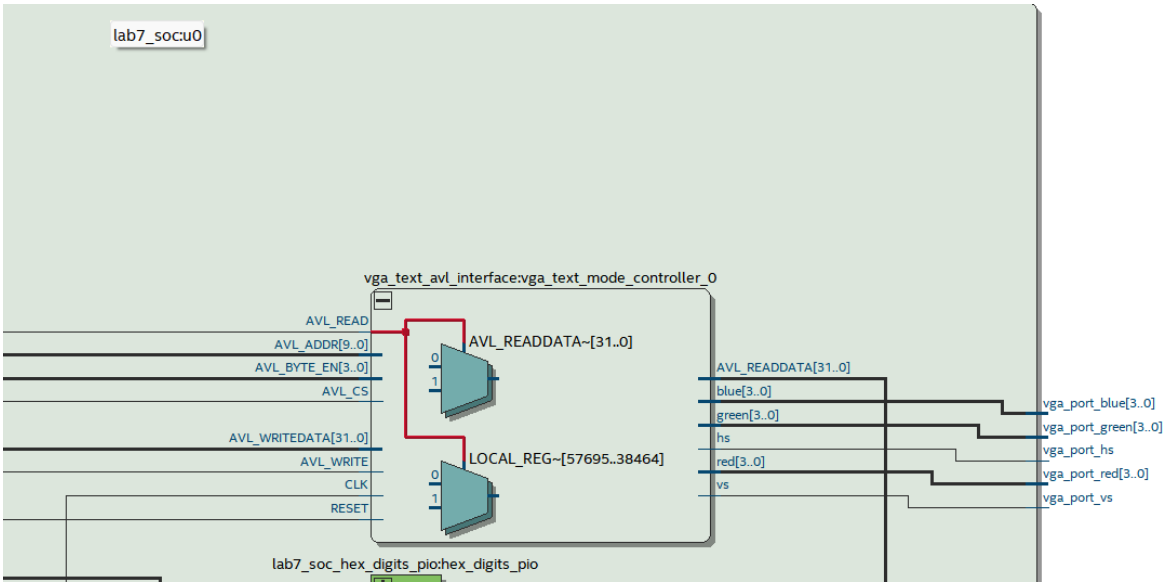


Figure 2: Week 1 VGA Controller Component

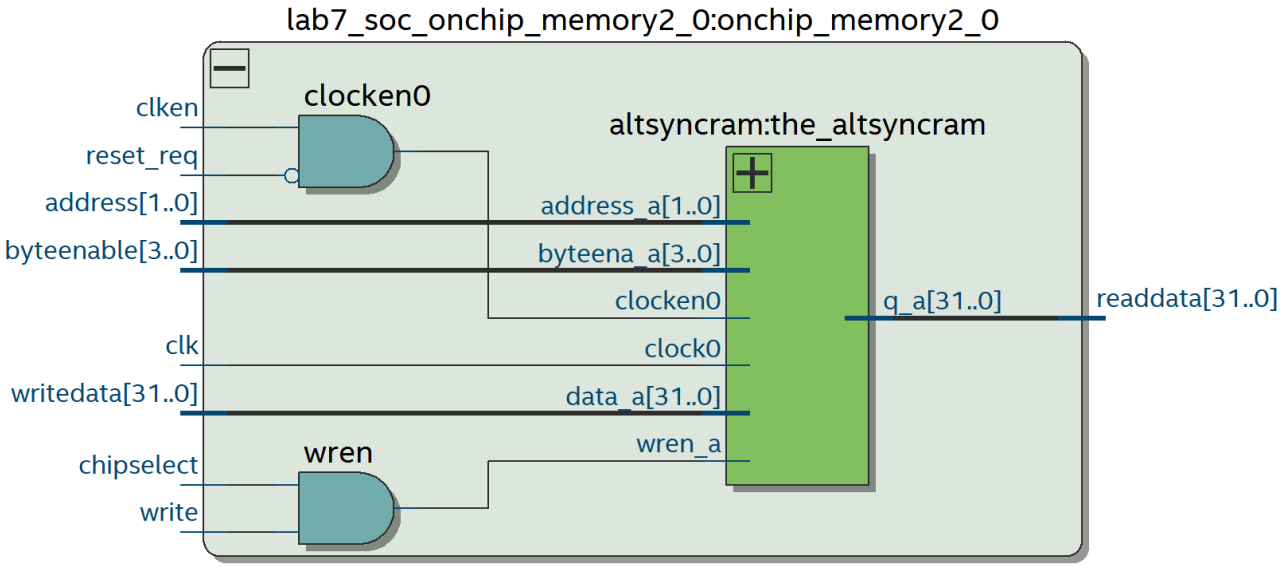


Figure 3: Week 2 On-Chip Memory

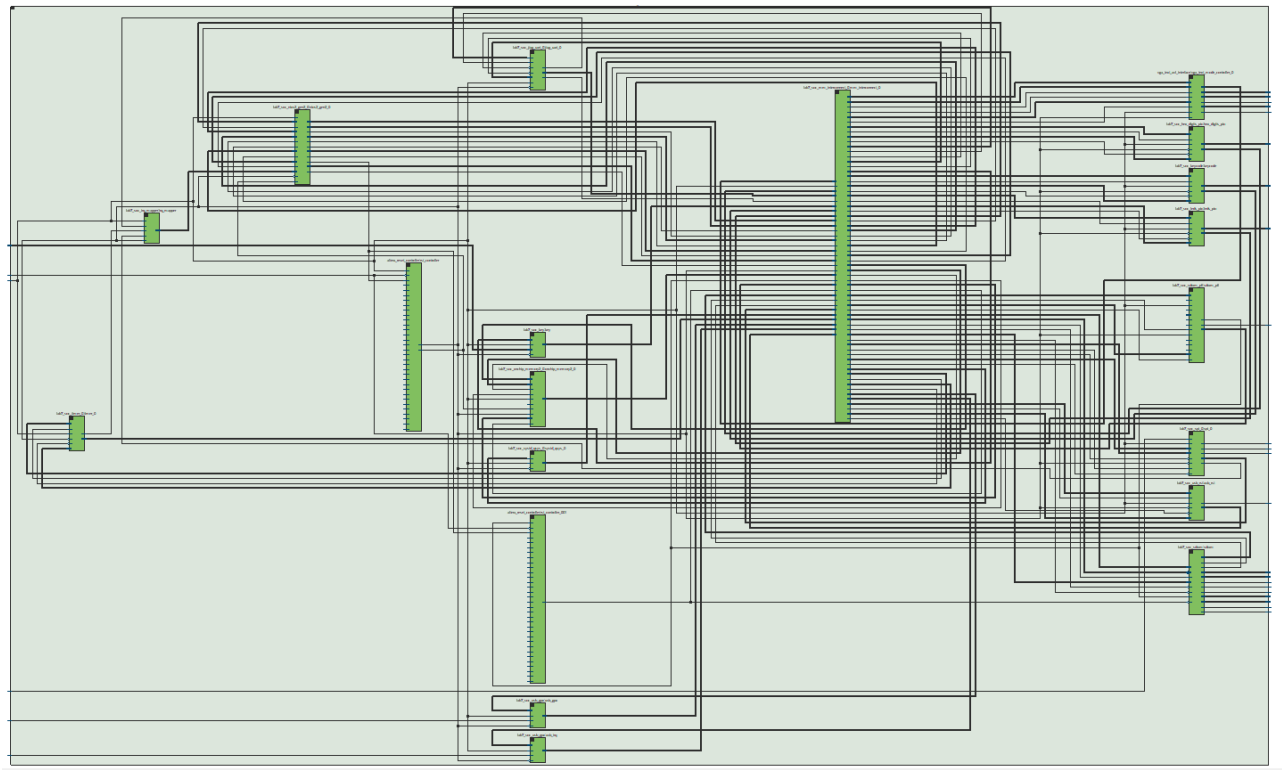
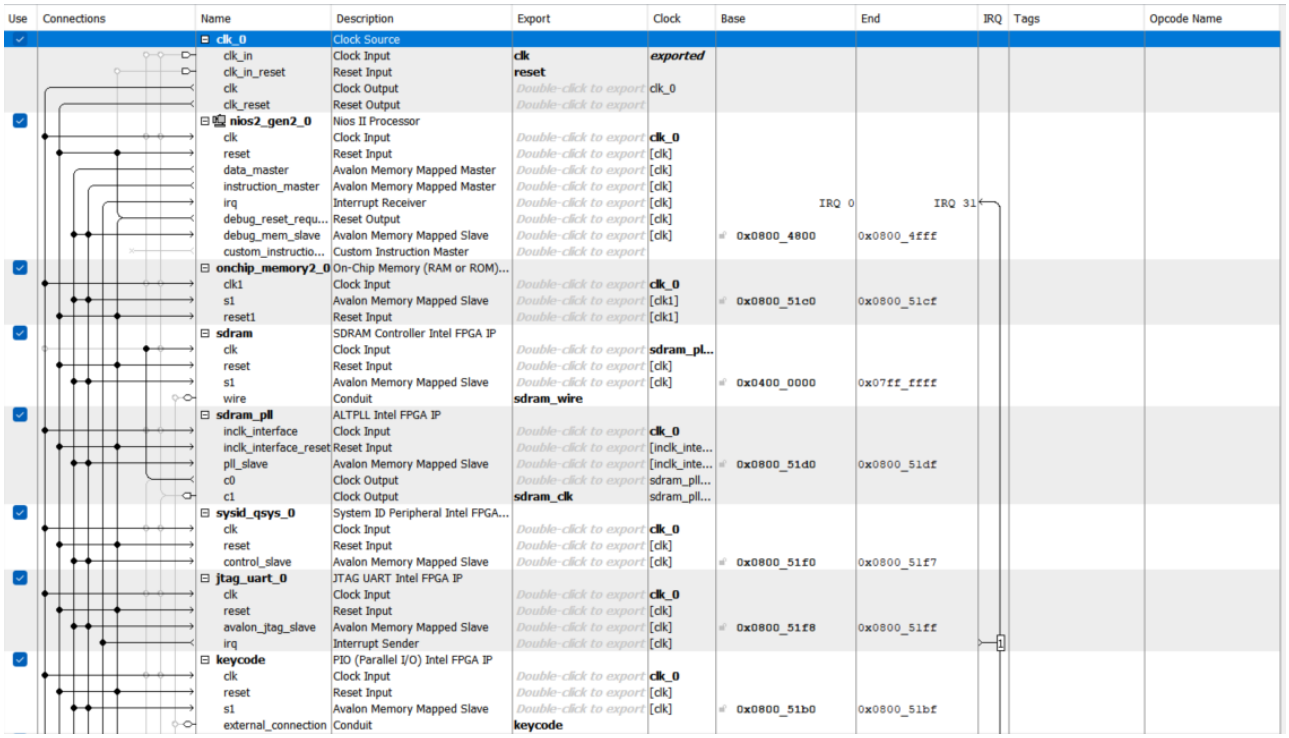


Figure 4: SoC Component for Week 1 & Week 2

Module Descriptions

Platform Designer



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	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 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*						
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)...							
		clk1	Clock Input	*Double-click to export*	**clk_0**					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0800_51c0	0x0800_51cf			
		reset1	Reset Input	*Double-click to export*	[clk1]					
☑		**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_51d0	0x0800_51df			
		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_51f0	0x0800_51f7			
☑		**jtag_uart_0**	JTAG UART Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_51f8	0x0800_51ff			
		irq	Interrupt Sender	*Double-click to export*	[clk]					
☑		**keycode**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	*Double-click to export*	**clk_0**					
		reset	Reset Input	*Double-click to export*	[clk]					
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_51b0	0x0800_51bf			
		external_connection	Conduit	**keycode**						

Figure 5: Platform Designer View (clk_0 to keycode)

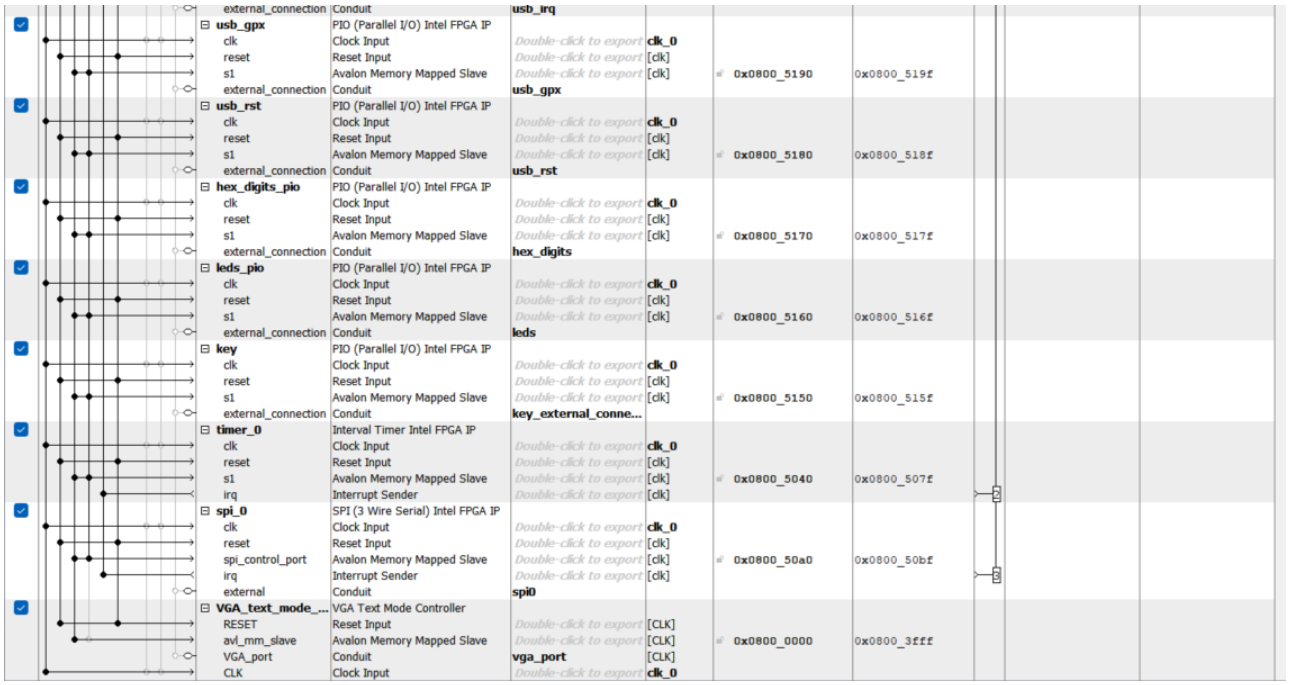


Figure 6: Platform Designer View (usb_gpx to VGA_text_mode_controller)

clk_0

This block is the Clock Source and generates the main clock that was used in our lab. It also produces the reset output which will be used throughout our lab when the `Reset` signal is connected. The clock and reset outputs are connected to all the other blocks so that the program can function in sync.

nios2_gen2_0

This is our NIOS II Processor block, and instantiates the processor we will use. It includes the master connections for data and instructions which are then driven to the slave peripherals used throughout our lab. Once configured with the clock and memory, it can be used to manage data communication between the different components of the lab.

onchip_memory2_0

This block instantiates the On-Chip Memory we need in the lab. It serves as one of the slave peripherals of the NIOS II processor so that it can pull data from the on-chip memory module.

sdram, sdram_pll

However our on-chip memory only has limited storage, so we also needed to instantiate an external off chip memory. The SDRAM Controller accomplishes this goal. Since the off-chip memory has complex accessing method, it cannot interact with the data bus directly. Therefore, we also declare the ALTPLL IP block which provides a separate clock signal for the sdram block. This separate timing method essentially allows our off-chip memory to interact with the Avalon bus, where the actual data transfer happens.

sysid_qsys_0

This block simply serves as a system ID checker, and its purpose is to ensure the compatibility between hardware and software. It prevents the user from accidently loading software onto the FPGA that has an older or incompatible NIOS II configuration.

jtag_uart_0

The JTAG UART peripheral allows for communication between the terminal of the host device and the NIOS II processor. It essentially allows us to use `printf` statements that we can use to debug our software while developing. However it is important to note that this is not considered good user interface because it requires a programming cable to be connected and Altera installed on the user's computer.

usb_irq, usb_gpx, usb_rst

These blocks were necessary on Platform Designer for proper connection to the USB chipset. The `irq` is the interrupt signal that is generated by the USB, and indicates a change in USB status, so whether or not it is plugged in. The `gpx` signal is used to control the power to a USB device. It controls an external power supply that provides additional power to the USB device in case the power provided by the port is not significant enough. The `rst` serves as a reset signal, when asserted, tells the device to return to its default or previous state.

keycode

This block is responsible for handling the keycodes that represent which key is pressed on the keyboard. The data is outputted to the user via the hex display on the FPGA.

hex_digits_pio, leds_pio

These function as output peripherals that get data from the Avalon bus when signaled by the master processor and output it using the displays on the FPGA.

key

Serves as an input for the program whose signal comes from the FPGA buttons. It communicates with the processor as another slave peripheral.

timer_0

This block provides a timer function that is used by the FPGA. It provides a timer that can be used by the rest of the program as a slave peripheral and allows for better communication and performance. It can generate periodic time signals and includes control signals for the timer.

spi_0

This is the SPI peripheral block is instantiated on platform designer to function with the rest of the program and the Avalon bus. This component also includes the interrupt signal and is integrated with the overall program so that all the other devices can communicate with one another.

VGA_text_mode_controller

This is a graphics controller IP that we created for this lab. The top level for this Platform Designer component is the `vga_text_avl_interface.sv`. It can support 80 columns by 30 rows, allowing for a total of 2400 characters. There are 128 available glyphs using 7 bits, including an additional 8th bit which can be set to draw the glyph with inverted colors. The bitmap for each character consists of 8x16 pixels, so the total screen resolution can be calculated as $80 * 8$ x $30 * 16$, resulting in a resolution of 640 by 480px. This graphics controller only supports text drawing, so each pixel is not individually accessible. Our controller also provides for 2.4k bytes of video memory (VRAM), which is directly mapped to the Avalon bus so that our NIOS II CPU can directly read and write to our memory. This allows our software C code to control the text being displayed on our monitor.

SV Modules

HexDriver.SV

```
module HexDriver (input  [3:0]  In0,
                  output logic [6:0]  Out0);
```

This is the HexDriver module that takes a 3-bit input and extends it to 7-bits so that it can output the corresponding hex digit. Its purpose is to assign the hex outputs that are displayed to the user on the FPGA.

vga_text_avl_interface.SV

```
module vga_text_avl_interface (
  // Avalon Clock Input, note this clock is also used for VGA, so this must be 50Mhz
  // We can put a clock divider here in the future to make this IP more generalizable
  input logic CLK,

  // Avalon Reset Input
  input logic RESET,

  // Avalon-MM Slave Signals
```

```
  input  logic AVL_READ,          // Avalon-MM Read
  input  logic AVL_WRITE,         // Avalon-MM Write
  input  logic AVL_CS,          // Avalon-MM Chip Select
  input  logic [3:0] AVL_BYTE_EN,     // Avalon-MM Byte Enable
  input  logic [11:0] AVL_ADDR,     // Avalon-MM Address
  input  logic [31:0] AVL_WRITEDATA,    // Avalon-MM Write Data
  output logic [31:0] AVL_READDATA,   // Avalon-MM Read Data

  // Exported Conduit (mapped to VGA port - make sure you export in Platform Designer)
  output logic [3:0]  red, green, blue, // VGA color channels (mapped to output pins in top-level)
  output logic hs, vs           // VGA HS/VS
);
```

The `vga_text_avl_interface` module takes the Avalon Slave Signals and clock and reset as inputs, and outputs the horizontal and vertical syncs along with the color channels so that the VGA can output to the monitors. The color channels are ultimately mapped to the pins in our top level, so that they can be seen on the monitor. The Avalon input signals are passed into are on chip memory instantiation so that our design can read and write data to memory. The purpose of this module is to instantiate the `font_rom`, `vga_controller`, and `ram` module so that we can get the character data from our `font_rom` data and use our color palettes to choose the colors to draw. Our on-chip memory stores our extended VRAM so that we can have shorter compilation time. This module also serves as our top level file for our graphics controller unit on Platform Designer.

VGA_controller.SV

```
module  vga_controller ( input        Clk,       // 50 MHz clock
                                      Reset,     // reset signal
                         output logic hs,        // Horizontal
sync pulse.  Active low
                              vs,        // Vertical sync pulse.
Active low
                          pixel_clk, // 25 MHz pixel clock
output
                          blank,     // Blanking interval
indicator.  Active low.
                          sync,      // Composite Sync signal.
Active low.  We don't use it in this lab,
                                     //   but the video DAC on
the DE2 board requires an input for it.
                 output [9:0] DrawX,     // horizontal
coordinate
                              DrawY );   // vertical coordinate
```

This is the VGA controller and connects all the VGA signals together so that the pixels can be outputted to the monitor or display. Based on the vertical and horizontal count the `DrawX` and `DrawY` signals are assigned so that the correct pixel can be drawn at the correct spot.

The purpose is to use the pixel beams to find the correct coordinates to draw the pixel.

lab7.SV

```
module lab7 (

      ///////// Clocks /////////
      input    MAX10_CLK1_50,

      ///////// KEY /////////
      input    [ 1: 0]   KEY,

      ///////// SW /////////
      input    [ 9: 0]   SW,

```

```
12      ///////// LEDR /////////
13      output   [ 9: 0]   LEDR,
14
15      ///////// HEX /////////
16      output   [ 7: 0]   HEX0,
17      output   [ 7: 0]   HEX1,
18      output   [ 7: 0]   HEX2,
19      output   [ 7: 0]   HEX3,
20      output   [ 7: 0]   HEX4,
21      output   [ 7: 0]   HEX5,
22
23      ///////// SDRAM /////////
24      output             DRAM_CLK,
25      output             DRAM_CKE,
26      output   [12: 0]   DRAM_ADDR,
27      output   [ 1: 0]   DRAM_BA,
28      inout    [15: 0]   DRAM_DQ,
29      output             DRAM_LDQM,
30      output             DRAM_UDQM,
31      output             DRAM_CS_N,
32      output             DRAM_WE_N,
33      output             DRAM_CAS_N,
34      output             DRAM_RAS_N,
35
36      ///////// VGA /////////
37      output             VGA_HS,
38      output             VGA_VS,
39      output   [ 3: 0]   VGA_R,
40      output   [ 3: 0]   VGA_G,
41      output   [ 3: 0]   VGA_B,
42
43
44
45
46
47      ///////// ARDUINO /////////
48      inout    [15: 0]   ARDUINO_IO,
49      inout              ARDUINO_RESET_N
50
```

```
51  );
```

`lab7.sv` is our top level module where we call all of our other VGA functions. All the output signals are labeled accordingly, and serve as inputs for our .soc module to instantiate all the blocks we created on Platform Designer. The VGA signals are used by VGA controller, and the outputs are assigned to the corresponding pixel color the is outputted by the `vga_text_avl_interface` module. The overall purpose is to bring all the other hardware components together so that the VGA can output a display to the screen. This module also uses the inputs to instantiate our SoC block.

font_rom.SV

```
1  module font_rom ( input [10:0]  addr,
2                    output [7:0]  data
3                  );
```

The `font_rom` module takes a 11-bit address input and outputs 8-bit data. The module stores all the possible characters that we can display on the screen. Each character is stored in 8x16px bit map, and the `font_rom` stores 128 different characters. The purpose of the module is to serve as the read only memory, and the address input is passed from the `vga_text_avl_interface` module. This address points to the start location of the bit map of the desired character, and the 7-bit output is then read by the logic to determine if each bit is 1 or 0, so that the `VGA_controller` can draw the correct pixel on the screen.

Software Component

For the software component of the lab, the code in week one was given to us. All we had to do was call the functions to run our demo tests. However for week two, we were given incomplete code and had to fill in some of the functions. The only file we changed was `text_mode_vga_color`, and the respective changes to the header file and the c file are described below.

text_mode_vga_color.h

```
struct TEXT_VGA_STRUCT {
  alt_u8 VRAM [ROWS*COLUMNS*2]; //Week 2 - extended VRAM
  //modify this by adding const bytes to skip to palette, or manually compute palette
  alt_u32 unusedBlock1[2047-1199];
  alt_u32 colorPalette[8];
};
```

In the `text_mode_vga_color.h` file, we had to fill out the `TEXT_VGA_STRCT` so that it represents our total memory for addressability. The extended VRAM was already allocated for us, however we had to create space for the 848 blocks of unused memory, and another 8 blocks for the color palette. The rest of the memory is also considered unused memory, but we did not specify it in the struct because the starting address for our second unused memory block is after our color palette, so we would not be accessing those memory addresses anyways since color palette was the only thing we need other then the VRAM.

text_mode_vga_color.c (shivam)

```
void setColorPalette (alt_u8 color, alt_u8 red, alt_u8 green, alt_u8 blue)
{
  //fill in this function to set the color palette starting at offset 0x0000 2000 (from base)
  if(color%2==1){
    alt_u32 mask = 0xfc003fff;
    vga_ctrl->colorPalette[color/2] &= mask;
    vga_ctrl->colorPalette[color/2] |= ((alt_u32)red)<<21;
    vga_ctrl->colorPalette[color/2] |= ((alt_u32)green)<<17;
    vga_ctrl->colorPalette[color/2] |= ((alt_u32)blue)<<13;
  }
  else{
    alt_u32 mask = 0xfff1e001;
    vga_ctrl->colorPalette[color/2] &= mask;
    vga_ctrl->colorPalette[color/2] |= ((alt_u32)red)<<9;
    vga_ctrl->colorPalette[color/2] |= ((alt_u32)green)<<5;
```

```
16      vga_ctrl->colorPalette[color/2] |= ((alt_u32)blue)<<1;
17    }
18  }
```

To support multicolored text, we had to add software code to draw our palette colors. We first had to edit our struct such that it followed the peripheral memory map we were given. We did this by creating an array called “unused” that was as large as the unused section (Unused spans from 0x4B0 - 0x7FF so it would be those two values subtracted from one another) and then the “color_palette” array of size 8 since we have 8 registers with two colors each. After this, we can change the setColorPalette function.

We had to use our given parameters to create our color palette so we could access and use it through our hardware code. We have eight registers with two colors each giving us 16 possible colors in our palette. We could use the color parameter to index through them and choose which one we want to utilize. If we divide color by 2, we can find out which register we want to choose and then we can mod color by 2 to find out if we are using the upper or lower RBG bits of our 32-bit color palette thus allowing us to choose one of the two colors in the register. To actually update the color now using the red, green, and blue parameter variables, we have to clear the according to RGB values. If our color%2 is 1 then we are using the upper RGB bits and we want to clear bits 24 through 13 otherwise we are using the lower RGB bits and we want to clear bits 12 through 1. To do this, we want and color palette with a mask that has 0s in place of the corresponding RGB bits that we want to clear and 1s everywhere else. Once we have done this, we can proceed to update those 12 bits with our red, green, and blue color bits by using the or function. We cannot do this, however, as the size of the colors (8 bits) does not match the 32-bit color palette and the colors are also in the wrong spot. To fix this, we cast red, green, and blue with alt_u32 to make each of them 32 bits. We then bit shifted red to the four bits of red that we cleared with the mask, and “or” the red value with the color palette. We repeat this with green and blue such that we have successfully cleared either the upper or lower RGB bits and updated them with the new RGB values. As we did all of this, we had to use vga_ctrl each time we made changes to the colorPalette. Therefore, rather than `colorPalette[color/2] |= ((alt_u32)red)<<21`, we would have `vga_ctrl -> colorPalette[color/2] |= ((alt_u32)red)<<21` so it can successfully change the colorPalette. After we did all of this, we could call on the colorPalette in the hardware code to assign a pixel’s color.

Design Resources and Statistics

7.1 Design Resources	
LUT	32488
DSP	0
Memory (BRAM)	11392
Flip-Flop	21775
Frequency	67.66 MHz
Static Power	97.35 mW
Dynamic Power	235.82 mW
Total Power	355.24 mW

Figure 7: 7.1 Design Resources and Statistics Table

7.2 Design Resources	
LUT	4911
DSP	0
Memory (BRAM)	129,024
Flip-Flop	2853
Frequency	71.53 MHz
Static Power	96.56 mW
Dynamic Power	69.60 mW
Total Power	188.22 mW

Figure 8: 7.2 Design Resources and Statistics Table

In each week of the lab, we used varying methods of storing memory. In week 1 of the lab, we created an array of registers in the SV code to be used locally in the code to complete our logic for drawing the sentences to the screen with the correct colors. In week 2, however, we were tasked with utilizing the FPGA on-chip memory such that we can choose our glyphs to then be drawn on the VGA monitor in one of the random 16 colors. The on-chip memory is used for storing larger amounts of data and instructions that need to be accessed frequently, while local registers are used for storing smaller amounts of data that are accessed frequently during processing. Registers are typically faster than on-chip memory but are also more limited in size. Because of this, using local registers would

be good to use in smaller labs but the more we use, the more we have to create and refresh every clock cycle, consequently increasing the compilation time. In week 1, we had compilation times between 10 and 15 minutes; however, in week 2 our compilation time was at a maximum of 5 minutes. Although using the registers used fewer resources the NIOS II processor had to keep accessing data from it reducing efficiency. In this situation, using OCM was much better as it allowed for more compilation and thus faster debugging. The efficiency of each implementation depends on the program as each of the two is good in its own way. We can further elaborate upon this when taking a look at our design resources and statistics between weeks 1 and 2. The difference in lookup tables alone is astounding as week 1 uses over an astounding 28,000 more lookup tables than the implementation in week 2. In the same respect, the BRAM usage in week 2 is significantly higher than week 1. Although each implementation has its own tradeoffs, we can see that the frequencies are quite similar so after compilation the speed of the actual functions are very close; however, using local registers requires almost twice the power that using OCM does and this shows how OCM is better for projects of a larger scale like this one.

Conclusion

Reflection on Design Functionality

For Week 1 of this lab, we had to spend a significant amount of time on how exactly we could access the memory to draw the characters we wanted on the screen. When we were initially doing the week 1 part of the lab, we were getting the wrong colors, and each of our individual characters was flipped. To fix this, we had to look into our color logic when we would draw on the screen and we found out that for every character that was to be drawn green, we were drawing blue and vice-versa. After finding this incredibly silly mistake we were getting the right colors but ran into the problem of flipping each of our glyphs. To do this, we changed the output of our Fontram, “fontdata,” from [7:0] to [0:7] so when we go to draw each row pixel by pixel we go backward. This fixed the problem. In week 2, we did not run into many issues but one of our big issues was getting the on-chip memory working with our week 1 logic. After a long time of debugging, we found that in our instantiation we used “vga_CLK” instead of “vga_Clk” and once we fixed this and hardcoded the color values for red, green, and blue, we could proceed to actually change the code for week 2. When we flashed to the screen, we found out that our colors were all wrong, and just like in week 1, we switched all green with blue and vice-versa so fixing this solved the color problem.

Potential Design Extensions

In this lab, we learned a very important skill: manipulating memory contents to draw something to the screen. We can use this for coloring sprites and drawing them to the VGA monitor. Additionally, we learned how to access a color palette to draw which is also incredibly useful when drawing sprites in our final project. We can extend this lab in the future for students to not choose a color for a sentence and draw it to the screen but to choose a sprite, create a color palette for it, and create logic for drawing it. This would significantly help for the final project.

Lab Manual Reflection

Any doubts we may have had after reading over the manual were resolved by attending lectures and watching past Q/A video streams as we were able to grasp a better visual understanding of the lab and how we can start implementing it. The manual provided us with knowledge of what was required of the design in the lab and what it must accomplish whereas the lectures and Q&A videos provided us with an understanding of how we may get started and with hints on designing our program. For next semester there is not anything that is unnecessarily difficult which can be improved upon for next semester.