ECE 385

Spring 2023

Lab 7

VGA Text Mode Controller with Avalon-MM Interface

Gustavo Fonseca, Hunter Baisden
TA: Hongshuo Zhang

Introduction

In this lab we are trying to create a text mode graphics controller that is able to print 80 columns of characters to the VGA Monitor. In Lab 7.1, we created a monochrome graphics controller that could only print black and white text. In Lab 7.2, we used a palette based approach that uses On Chip Memory (OCM), allowing us to draw text in a wide range of colors.

Our design built on our Lab 6 design by using the same logic to control the VGA output, VGA_controller.sv. We also built upon the same Platform Designer that we had developed for Lab 6 as well, ensuring that the connections we established with the VGA output and the Microcontroller continued to work.

Written Descriptions

1. Week 1

a. Lab 7 System

The Lab 7 System worked through connecting the VGA controlling Top Level, vga_text_avl_interface.sv, and the Avalon-MM Slave. The 32-Bit Avalon MM Bus would send the current characters to be printed onto the screen. This data would then be saved within the correct register based on the address sent by the Bus. The vga_text_avl_interface.sv would then use the character algorithm to know where to print the characters saved within the registers. The module would then pull the correct font for the desired character from font_rom.sv. Finally, the VGA_controller.sv would then receive the necessary signals to output the correct characters to the screen.

b. VGA Controller High Level

The vga_text_avl_interface.sv is the Top Level of the design. As inputs it takes in Clock, Reset, the Avalon Signals (Read, Write, Chip Select, Address, etc.). The only outputs are directly to the VGA which are; Red, Green, Blue (RGB), Vertical Sync (VS), and Horizontal Sync (HS). The Controller encapsulates 600 Registers to store characters (VRAM) and one Control Register which controlled the color of the text. If the Avalon Bus is writing data, then the write data will be stored within the correct register based on the sent address. If the Bus is reading data, then the correct register based on the sent address will be stored within the read data signal that will be sent back to the Avalon Bus as it is bidirectional. The Controller then goes through the character algorithm to determine what position and what character should be printed to the screen given the current position of the Virtual Electron Beam. The current character address is then sent to the font ROM and the corresponding data is sent out of the ROM and is then printed on the current pixel. The color of the pixel is determined by the foreground and background colors set by the Control Register. The pixel can also be inverted if the inverse color bit is high. Finally, the RGB values are output to the VGA.

c. VGA Register Logic

The way we read and write to the VGA Registers is interfacing with the Avalon Bus Signals. If the reset button is pressed, then all the registers will be set to a value of 0. If the Avalon Chip Select is high, then that means that we are either going to read or write into the registers. If Avalon Write Enable is high, then depending on the Avalon Byte Enable, a certain portion of the Avalon Write Data will be stored into the register selected by the Avalon Address. If Avalon Read Enable is high, then the 32 bits of data from Avalon Read Data will be stored into the register selected by the Avalon Address.

d. Text Algorithm

The algorithm to draw the text characters is as follows:

i. Divide the current X position (cx) by 8 and the current Y position (cy) by 16 to find the current X and Y index. This division accounts for the size of each character.
ii. Find the current 2D index with the equation: ((cy * 80) + cx) / 4.
iii. Define the offset of the current character by the 2 least significant bits of current X.
iv. Based on the offset we define which one of the 4 characters stored within the register selected by the current 2D index. We then save the selected character in char.
v. In order to find the font address we use the following formula: (char / 16) + [3:0] current Y position.
vi. Get the output from the Font ROM based on the font address, rom_out. We then take the negative value of [2:0] current X position, -drawxl[2:0]. We then take the value of rom_out at the "-drawxl[2:0]" index to get the final output to the VGA.

e. Inverse Color Bit

The way we implemented the inverse color bit was by using an XOR between the bit and the output bit. If the XOR's output is high, the RGB output will be current Foreground RGB Value. If the XOR's output is low, the RGB output will be the current Background RGB Value.

The way we implemented the Control Register is by defining it as a 601st register. We then read and write from it like any other register. We then define the foreground and background RGB values based on different sections of the register.

2. Week 2

a. VRAM Modifications

The main modifications when changing from registers to On Chip Memory (OCM) is the interaction with reading and writing to the OCM. The OCM takes the Avalon Bus Write Data as its data in. The write address is connected to the Avalon Bus Address. The Write enable bit is defined by the formula: Avalon

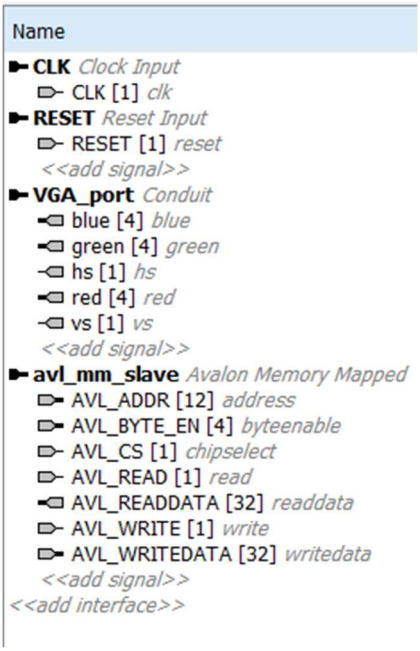
Write Enable & Avalon Chip Select & !(Most significant bit of Avalon Address). The read address is defined by the current 2D character index. Finally, the output, q, is the actual line of characters we will print to the screen.
We designed our OCM to be a one port system with a bidirectional port. With this, we were able to write and read to the OCM through one port. Our design shared this one port by making sure the write and read enable were not the high at the same time.

```
ocm3 ocm3 (
                .byteena_a(AVL_BYTE_EN),
                .clock(CLK),
                .data(AVL_WRITEDATA),
                .rdaddress(index),
                .rden(1'b0),
                .wraddress(AVL_ADDR[10:0]),
                .wren(AVL_WRITE & AVL_CS & !AVL_ADDR[11]),
                .q(wdata)
                );
```

b. Platform Designer Modifications

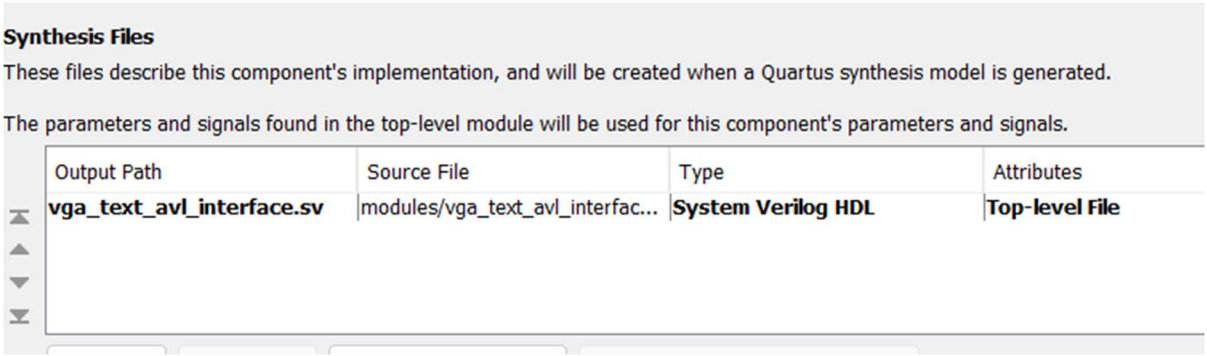
The platform designer remained mostly the same from lab 6 except for the addition of the custom IP for Lab 7. We created a custom text graphics memory controller that is connected to the Avalon memory mapped bus to create a VGA output that is 80 columns long.

Some additional notes about the Custom IP: It is built in the following structure:



The IP takes in a CLK and RESET signal that are connect to the rest of the NIOS II. In order to interact with the memory, the avl_mm_slave signal is used to interact with the on-chip memory of the FPGA. The VGA_port conduit is then used to output the three colors red, green, and blue as well as the hs and vs signals.

To link this custom IP to the synthesis of the design we designed a file called vga_text_avl_interface.sv and linked it to the custom IP below.



Instantiating and running this custom IP is how we are able to have more advanced graphics on the FPGA.

i. **Sprite Algorithm Modifications**

The modified algorithm goes as follows:

Divide the current X position (cx) by 8 and the current Y position (cy) by

16 to find the current X and Y index. This division accounts for the size of each character.

ii. Find the current 2D index with the equation: ((cy * 80) + cx) / 2. We had to modify this from dividing by 4 to 2 as there is half as many rows.

iii. Define the offset of the current character by the least significant bit of current X. The offset changed from 2 bits to 1 bit because there is only 2 characters within each memory location now instead of 4.

iv. Based on the offset we define which one of the 2 characters stored within the register selected by the current 2D index. We then save the selected character in char, the front index into front_idx, the back index into back_idx, and the inverse bit into iv. We need the front and back indexes now because the palette needs them for the foreground color and the background color.

v. We then divide both front_idx and back_idx by 2 to get the correct form to output to the VGA. We store these values in front_front and back_back respectively.

vi. We define the foreground and background based on the LSB of the front_idx and back_idx, storing the palette selected by the front_front and back_back respectively.

vii. In order to find the font address, we use the following formula: (char / 16) + [3:0] current Y position.

viii. Get the output from the Font ROM based on the font address, rom_out. We then take the negative value of [2:0] current X position, -drawxl[2:0]. We then take the value of rom_out at the "-drawxl[2:0]" index to get the final output to the VGA.

c. Multicolored Text Modifications

The main modification that was done to support multicolored text was the inclusion of a foreground index and a background index. These indexes select which palette will be used to color the foreground and background of that specific character. We had to modify how we took in the specific character. The 32-bit value is divided into 2 characters now instead of 4. Thus, each character has 16-bits characterizing it: [3:0] background index, [7:4] foreground index, [14:8] the specific character, [15] inverse color bit. The first bit of the foreground and background index determines whether to use the bits [12:1] from the selected palette or the [24:13] from the selected palette. This separation of the foreground and background allows for the different colors on the screen at once.

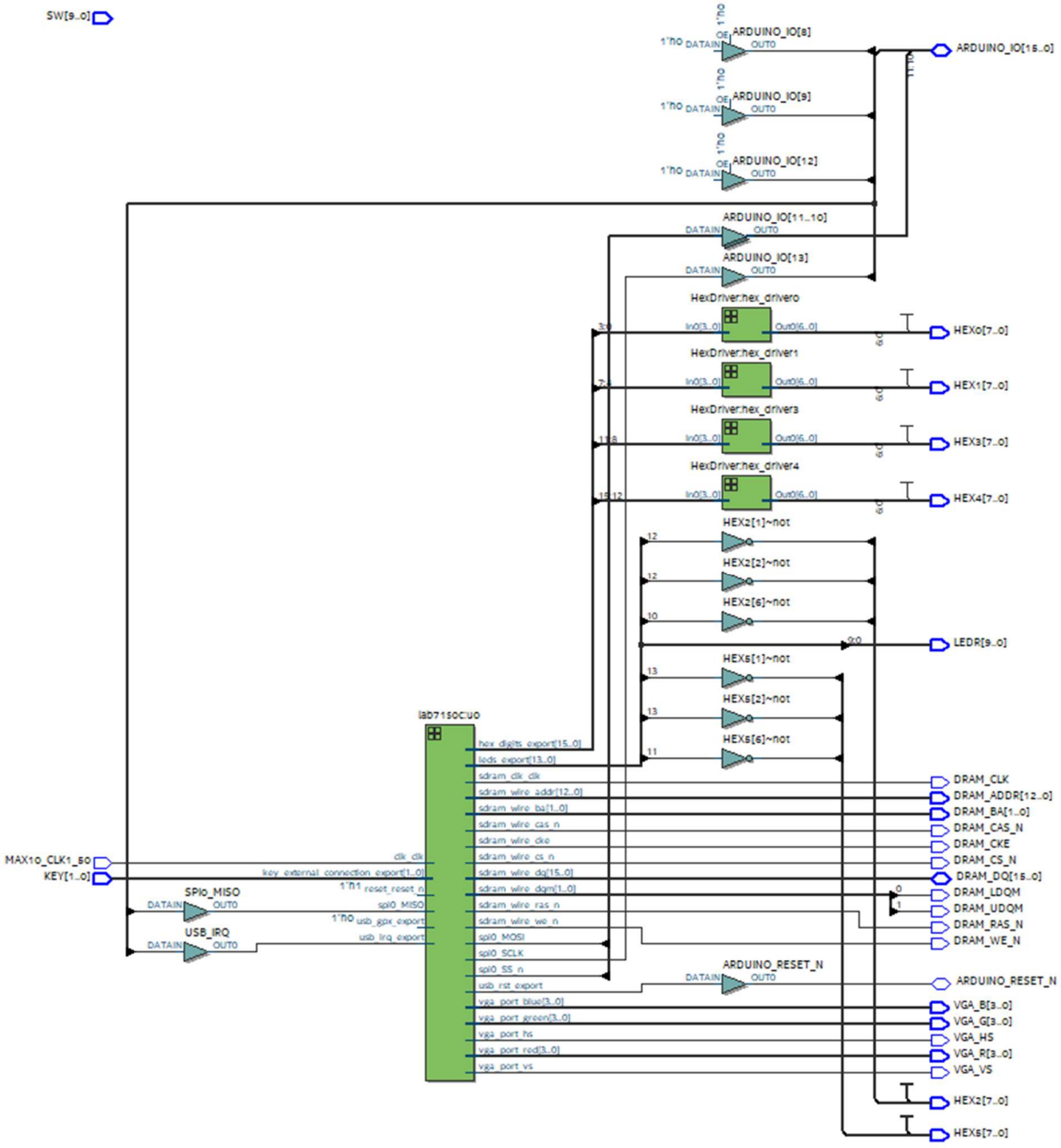
d. Palette Color Code

Instead of going for the traditional struct approach to the set color palette function we chose to use a dynamic pointer. The code is seen below:

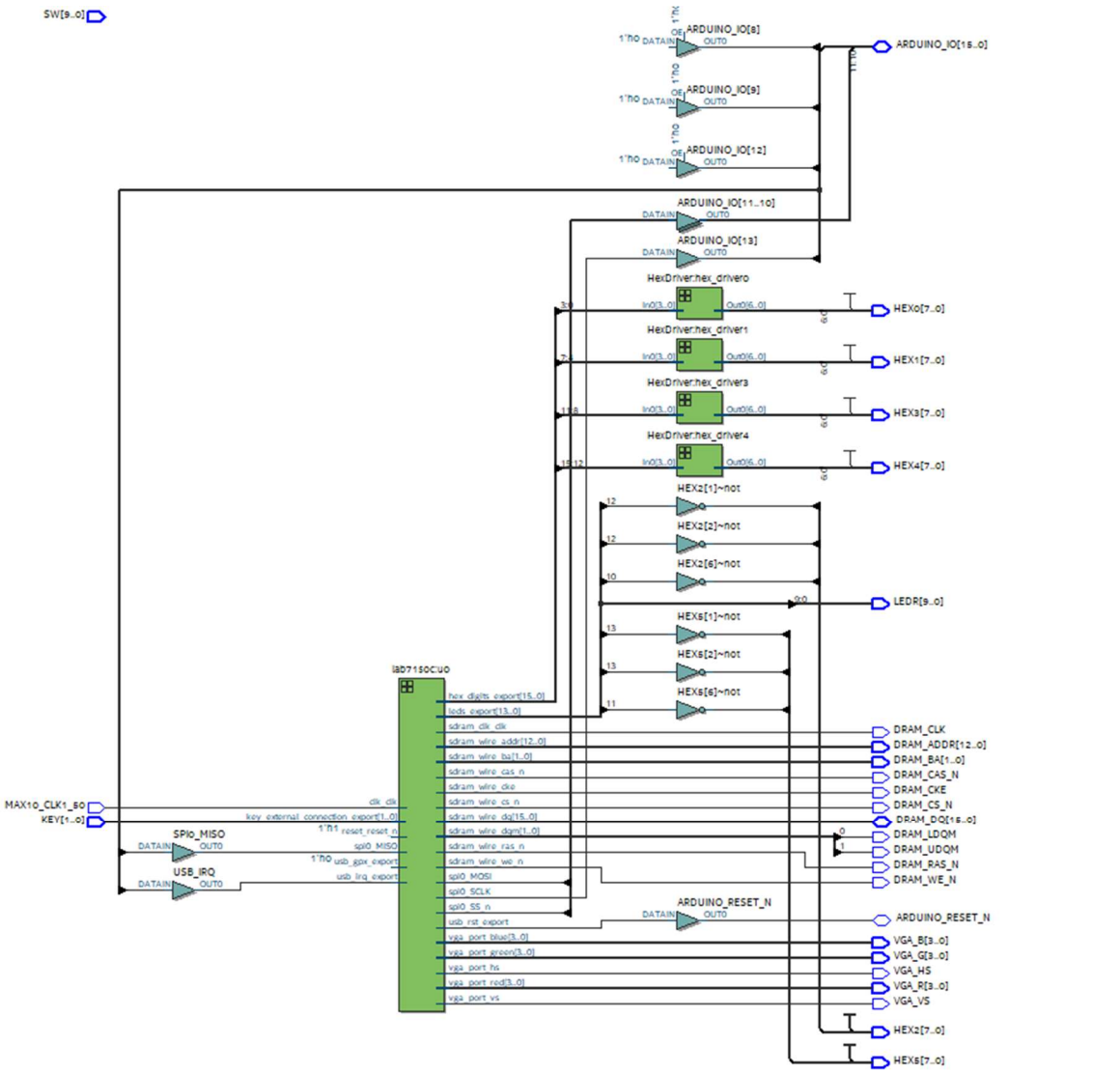
```
void setColorPalette (alt_u8 color, alt_u8 red, alt_u8 green,
alt_u8 blue){

alt_u8 color_index = color / 2;
alt_u32*address =(alt_u32 *)(vga_ctrl);
address = address + color_index + 2048;// offsetting the
pointer to the proper spot
alt_u32 temp = 0x0;
alt_u8 color_set = color % 2;
if (color_set == 0){
    *address = 0;
        temp = (blue << 1) + (green << 5) + (red << 9);

}
else if (color_set == 1) {
    temp = (blue << 13) + (green << 17) + (red << 21);

}
*address = temp + *address;
}
```

We set the color index to color /2 and then set the address to the vga control address. We then added a 2048 offset to account for the size of the memory. Finally we calculate the color_set and check the value to adjust the address of the color pallette.

Block Diagram



Above is the block diagram for week one of the lab.



Above is the block diagram for week two of the lab. It should be noted that the SOC is different, we just used the same name to make things easier.

Module Descriptions

Module: lab7 (in lab7.sv)

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

Outputs: [9:0] LEDR, [7:0] HEX (0-5), 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, [3:0] VGA_G, [3:0] VGA_B

Inouts: [15:0] DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N

Description: This is the top-level module for this project. It declares the SOC as well as the the necessary Arduino components.

Purpose: This module serves as the top level for all the interactions with the FPGA. It is used as the top level in compilation when we program the FPGA. A diagram for this top level can be seen above for both part 1 and 2.

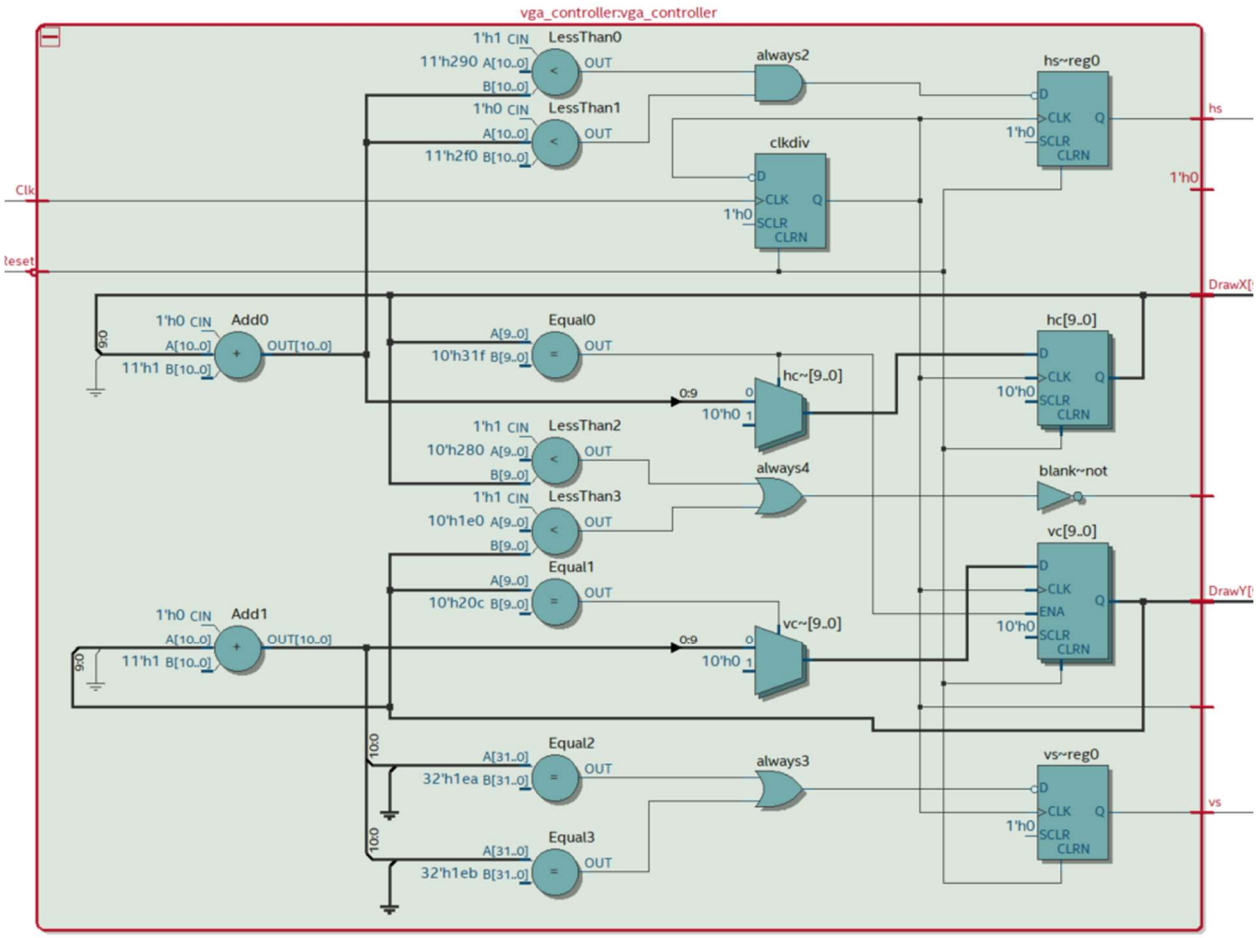
Module: vga_controller (in vga_controller.sv)

Inputs: Clk, Reset

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

Description: This module controls the timings and locations on the vga output itself. It simply takes in the Clk and Reset and outputs the vsync and hsync, as well as the pixel_clk, sync, blank, and the coordinates DrawX and DrawY.

Purpose: This module is important to make sure the vga output is properly timed and also not blurring and displaying incorrect pixels.



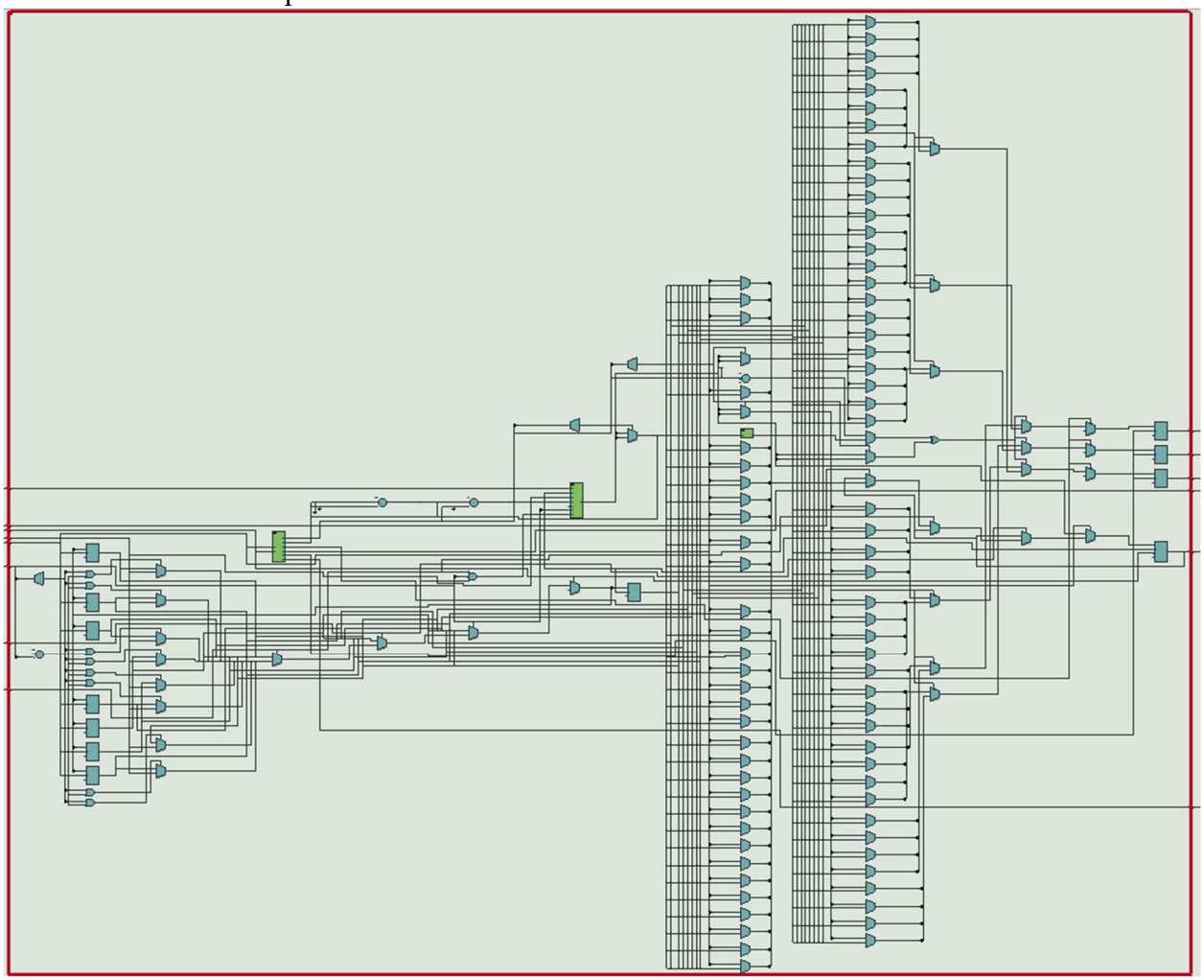
Module: vga_text_avl_interface (in vga_text_avl_interface.sv)

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 is used in both sections of the lab as a more advanced color mapper. It uses a custom SOC component that we made in the platform designer.

Purpose: Without this module we would not have complete color control over the entire screen, and the colors on each part of the screen.



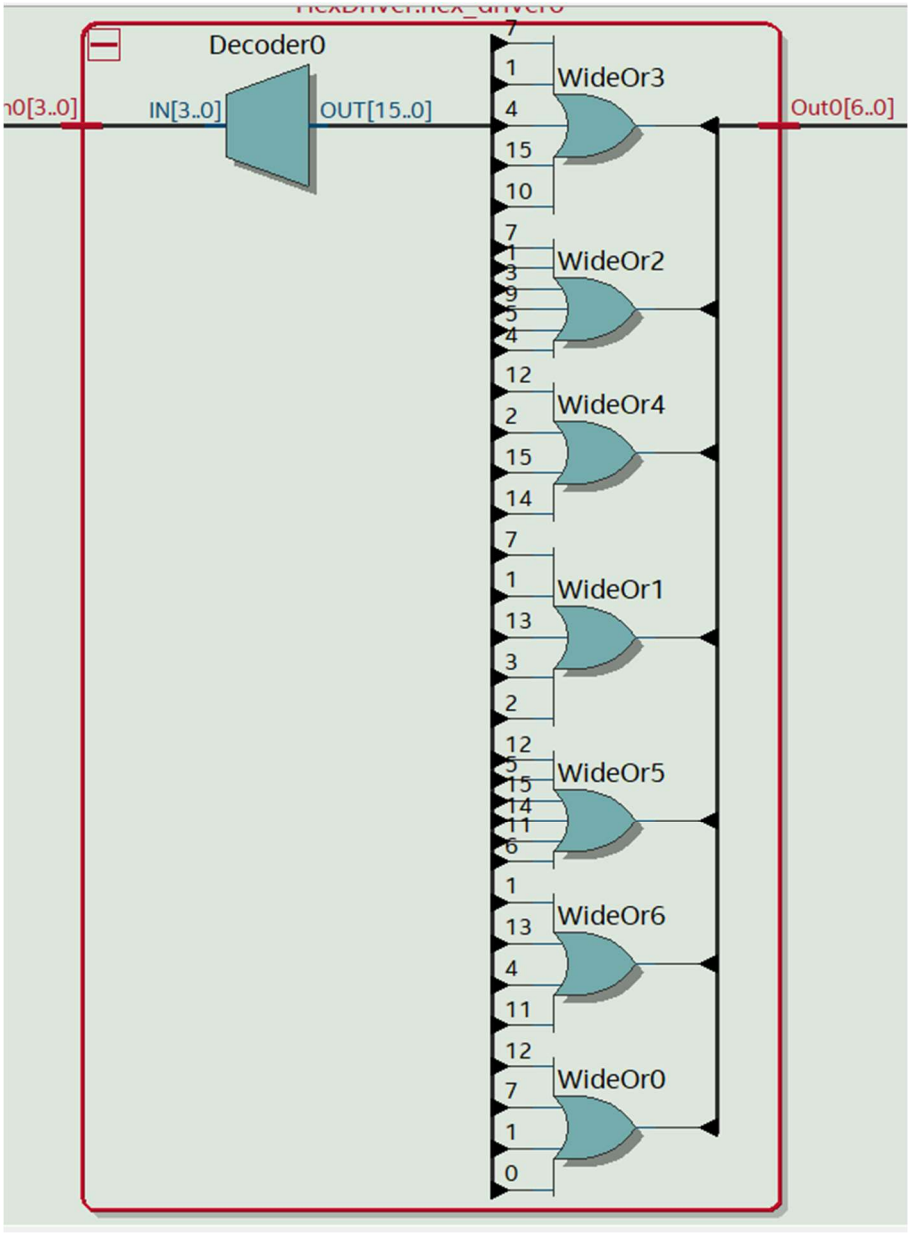
Module: HexDriver (in HexDriver.sv)

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This module pairs every unique case of 4-bits (16 in total) into a 7-bit value used to illuminate the hex displays.

Purpose: This was used to output the values we were working with to the hex displays, both for debugging and for the demo itself.



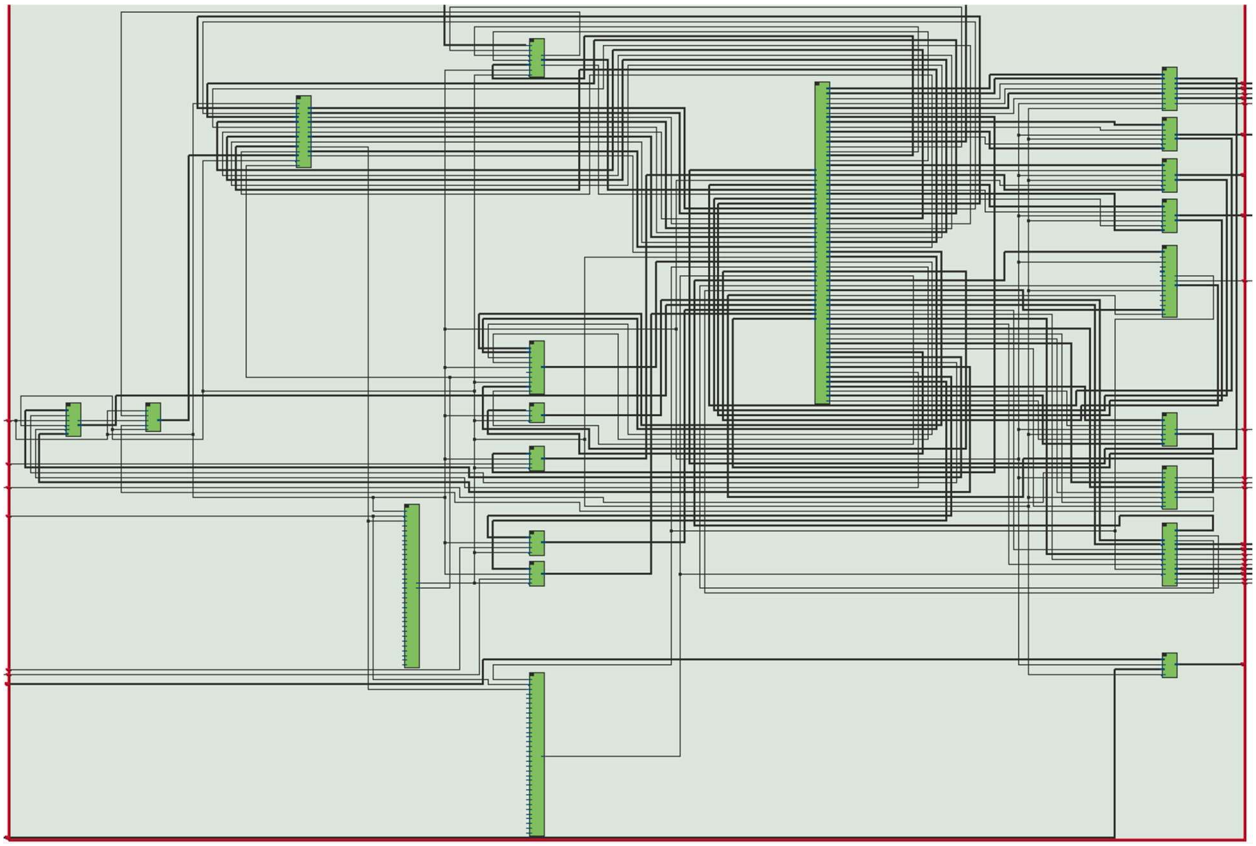
Modules: lab71soc (in lab71soc.qip)

Description: Below I will describe the PIO blocks added in addition to the ones needed for the NIOS II processor to complete the platform designer.

Purpose: All these modules come together to make a working NIOS II processor with memory USB, and VGA.

System Level Block Diagram

If not explicitly mentioned, the SOC component was used in both parts of the lab. An overview of the SOC is shown below:



The rest of this section goes into further detail about the different IP's within the platform designer, generated within the SOC.

Module: nios2_gen2_0 (in lab71soc.qip)

Connections: clk, reset, data_master, instruction_master, irq, debug_reset, debug_mem_slave

Exports: None

Description: This is the central processor that everything is connected to. It is a NIOS II Processor from Intel.

Purpose: This module is what everything else is connected to. The block diagram for this module is far to large to fit into a page.

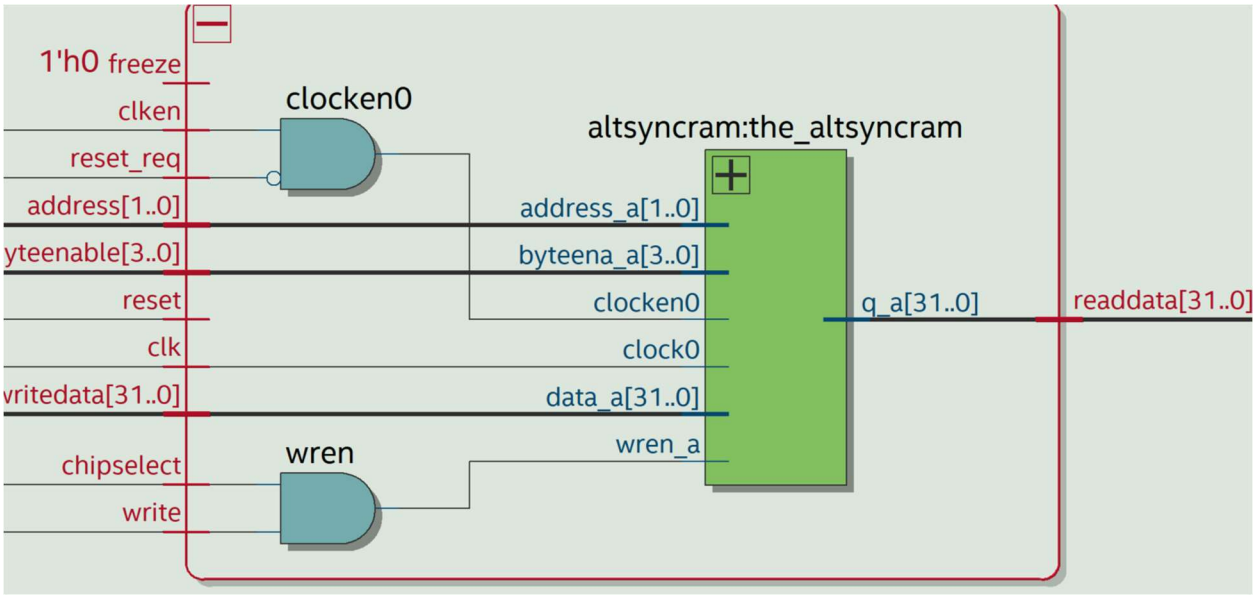
Module: onchip_memory2_0 (in lab71soc.qip)

Connections: clk1, reset1, Nios II Data bus

Exports: None

Description: This is an On chip memory module used for the on chip memory on the FPGA

Purpose: This module is not required, but makes storage easier



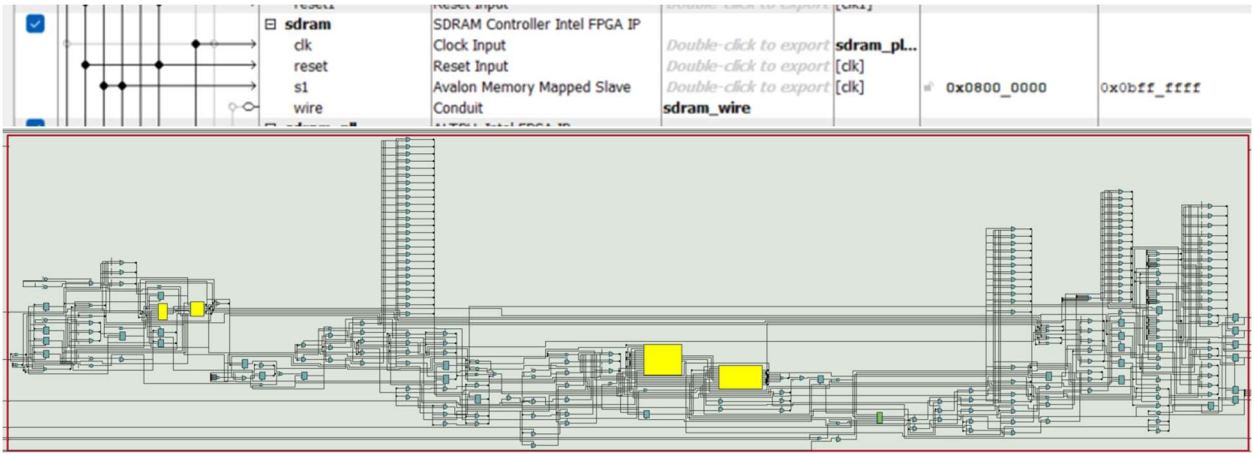
Module: sdram (in lab71soc.qip)

Connections: clk1, reset1, Nios II Data bus

Exports: sdram_wire

Description: This is the functional synchronous memory for the NIOS II processor.

Purpose: Without this, the processor would not be able to store any data.



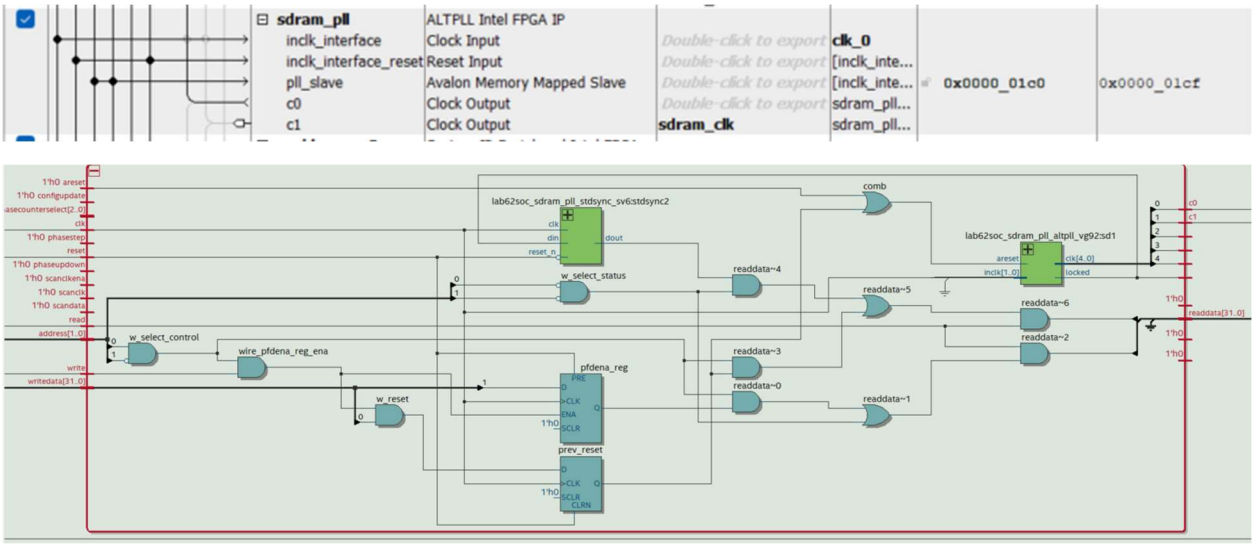
Module: sdram_pll (in lab71soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: sdram_clk

Description: This is the module that is used to control the SDRAM, specifically clocking it at the right frequency.

Purpose: Without this module, the SDRAM would not work well, if at all, because of asynchronous clocking times.



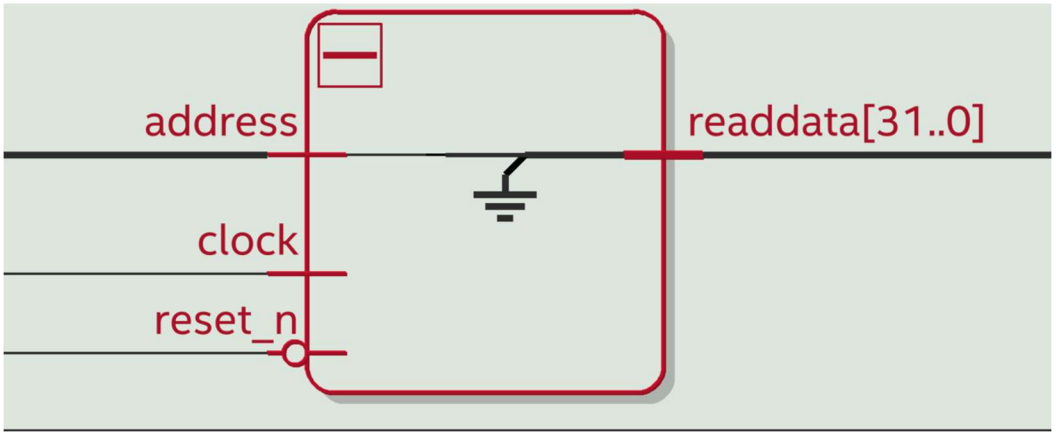
Module: sysid_qsys_0 (in lab71soc.qip)

Connections: clk, reset, control_slave

Exports: None

Description: This module acts as a manager between the hardware and the BSP that is generated.

Purpose: Without this module, the BSP would not work, and in turn the c code could not be run on the FPGA .



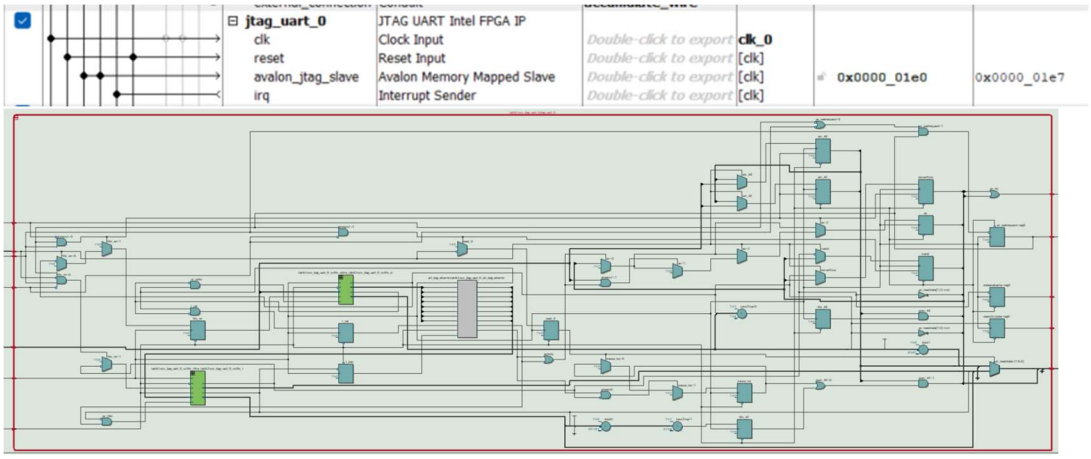
Module: JTAG UART Intel FPGA(in lab71soc.qip)

Connections: clk, reset, Nios II Data bus, IRQ

Exports: None

Description: This is a module within the platform design that is specifically used for USB connections. This was for Part 2 of the lab.

Purpose: This allows for the printf() command to be used in the eclipse IDE.



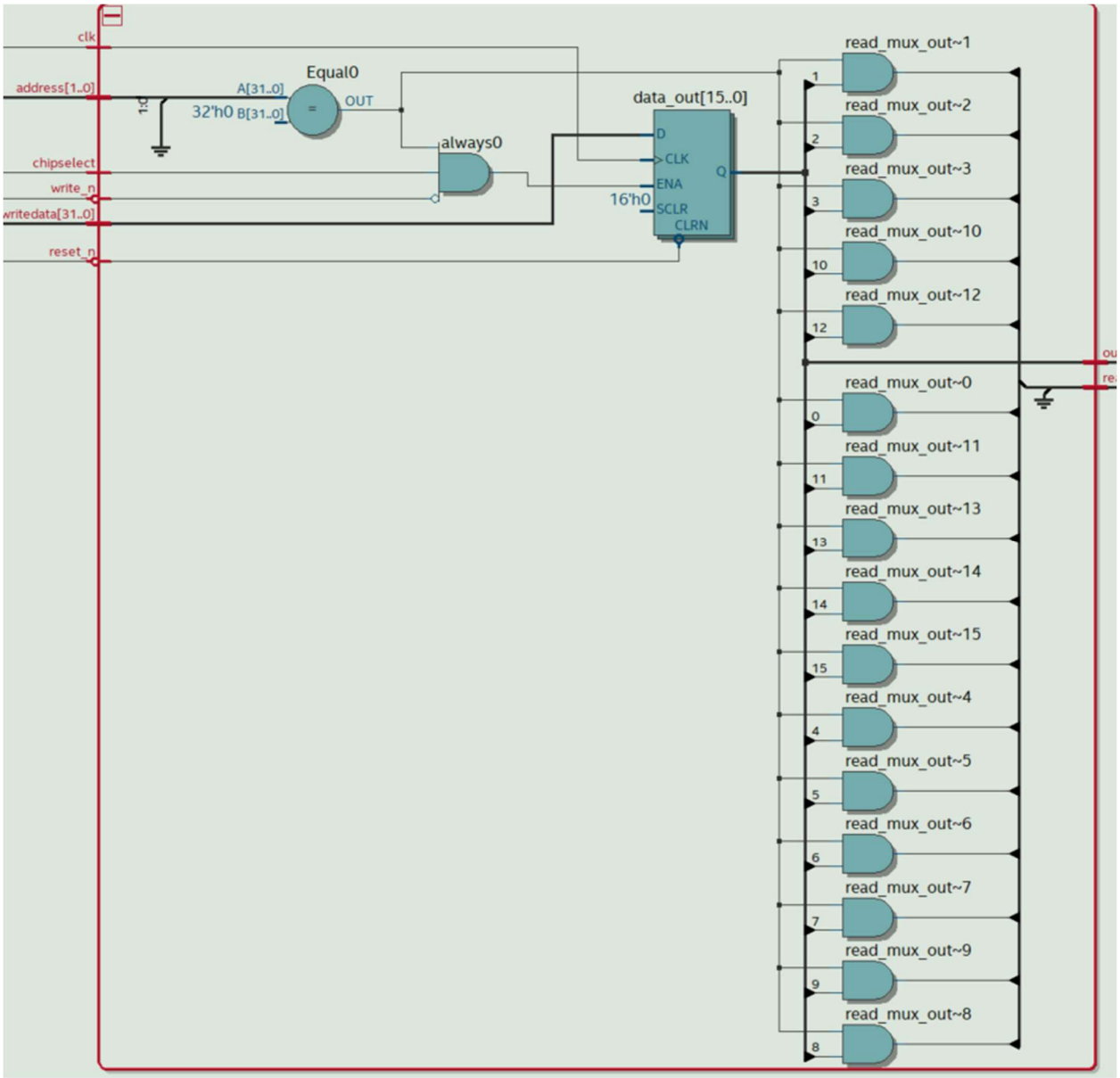
Module: hex_digits_pio (in lab71soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: hex_digits

Description: This is the PIO used to print values to the hex displays coming from the keyboard.

Purpose: This PIO is required to get the hex from the keycodes to the Hex Drivers.



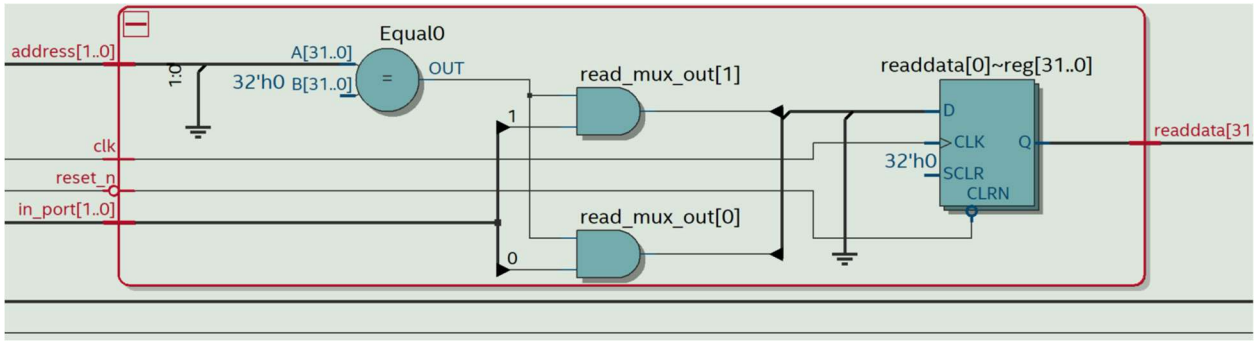
Module: key (in lab71soc.qip)

Connections: clk, reset, Nios II Data bus

Exports: key_s1, key_external_connection

Description: This is the PIO used to detect key presses on the FPGA. Both the key[0] and key[1] are assigned to different functions/keys on the FPGA.

Purpose: This module was required to make sure the 2 buttons were usable particularly to be able to reset the screensaver



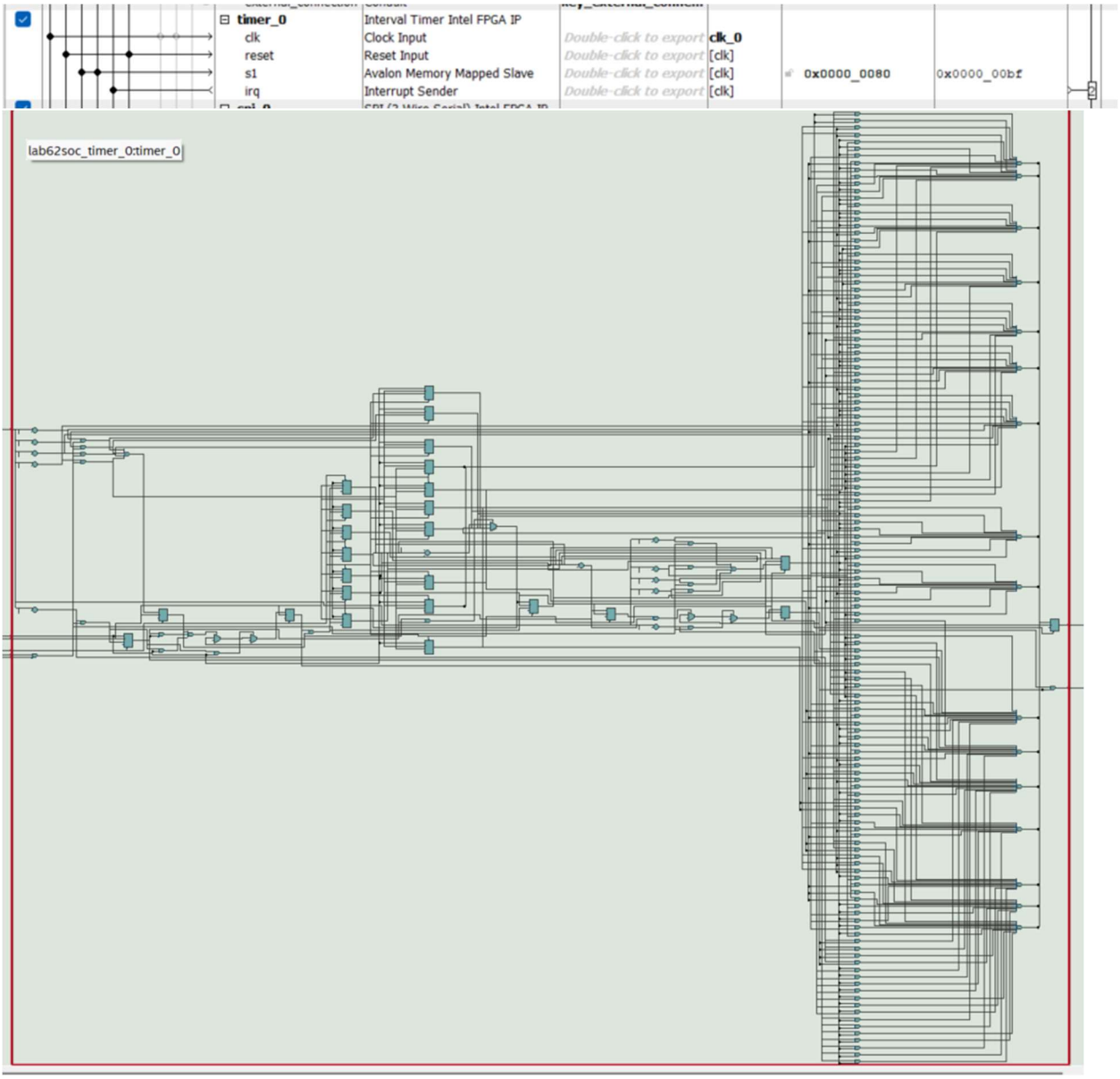
Module: timer_0 (in lab71soc.qip)

Connections: clk, reset, Nios II Data bus, IRQ

Exports: None

Description: This Intel timer IP is used to keep the timings in order. It is required for USB timings to be exact.

Purpose: This time is used to line up the timings with the IRQ as well as with the SPI.



Module: spi_0 (in lab71soc.qip)

Connections: clk, reset, Nios II Data bus, IRQ

Exports: spi0

Description: This is a SPI(3 Wire Serial) Intel IP and is responsible for the majority of the driver work between the USB and the NIOS II. This was added for part 2 of the lab.

Purpose: This IP is the powerhouse between the NIOS II and the USB, and without it, a connection between the 2 would be infeasible. The block diagram for this section is far too large and complicated to fit into this lab report.

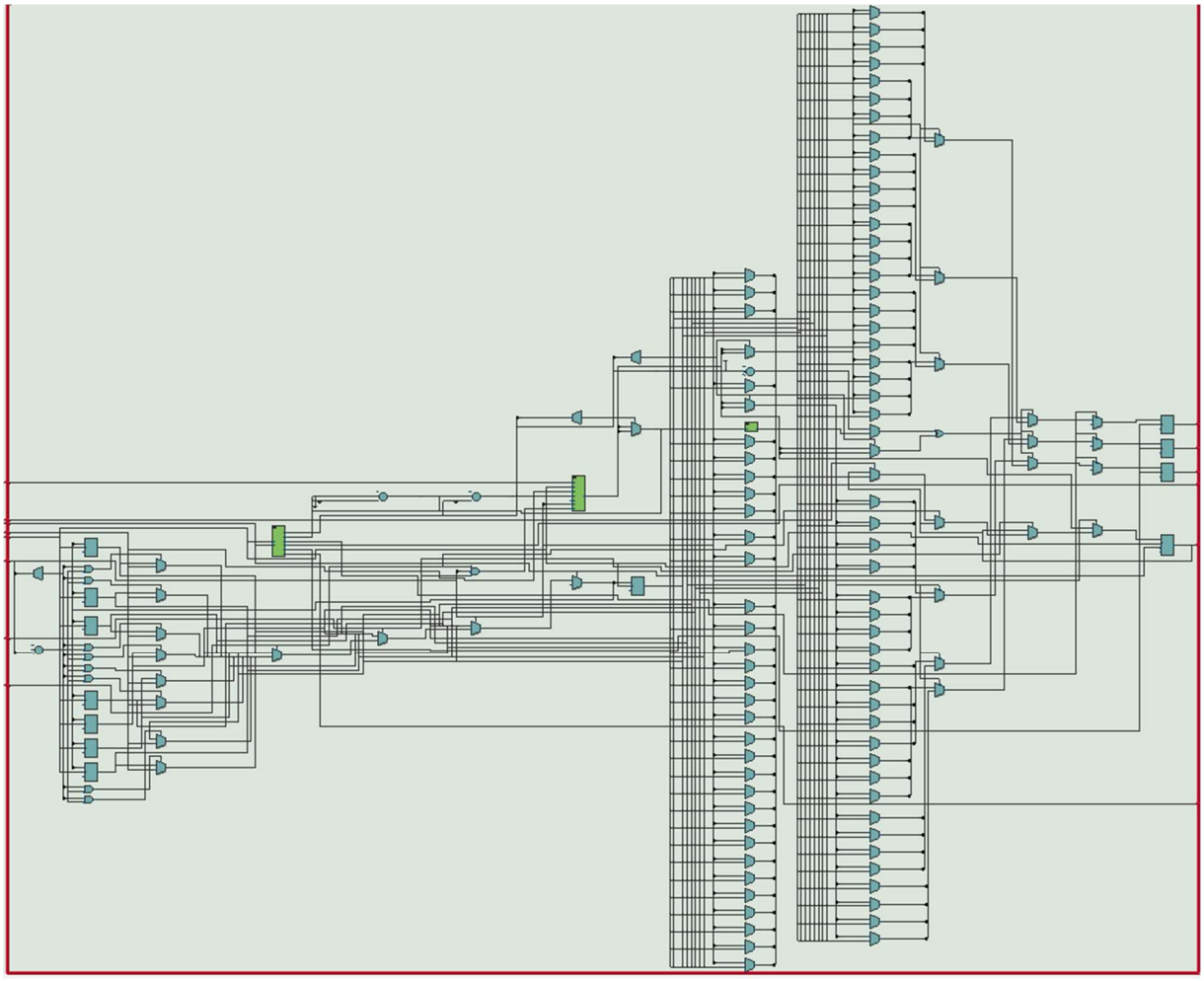
Module: VGA_text_mode_controller_0 (in lab71soc.qip)

Connections: CLK, RESET, avl_mm_slave, VGA_port

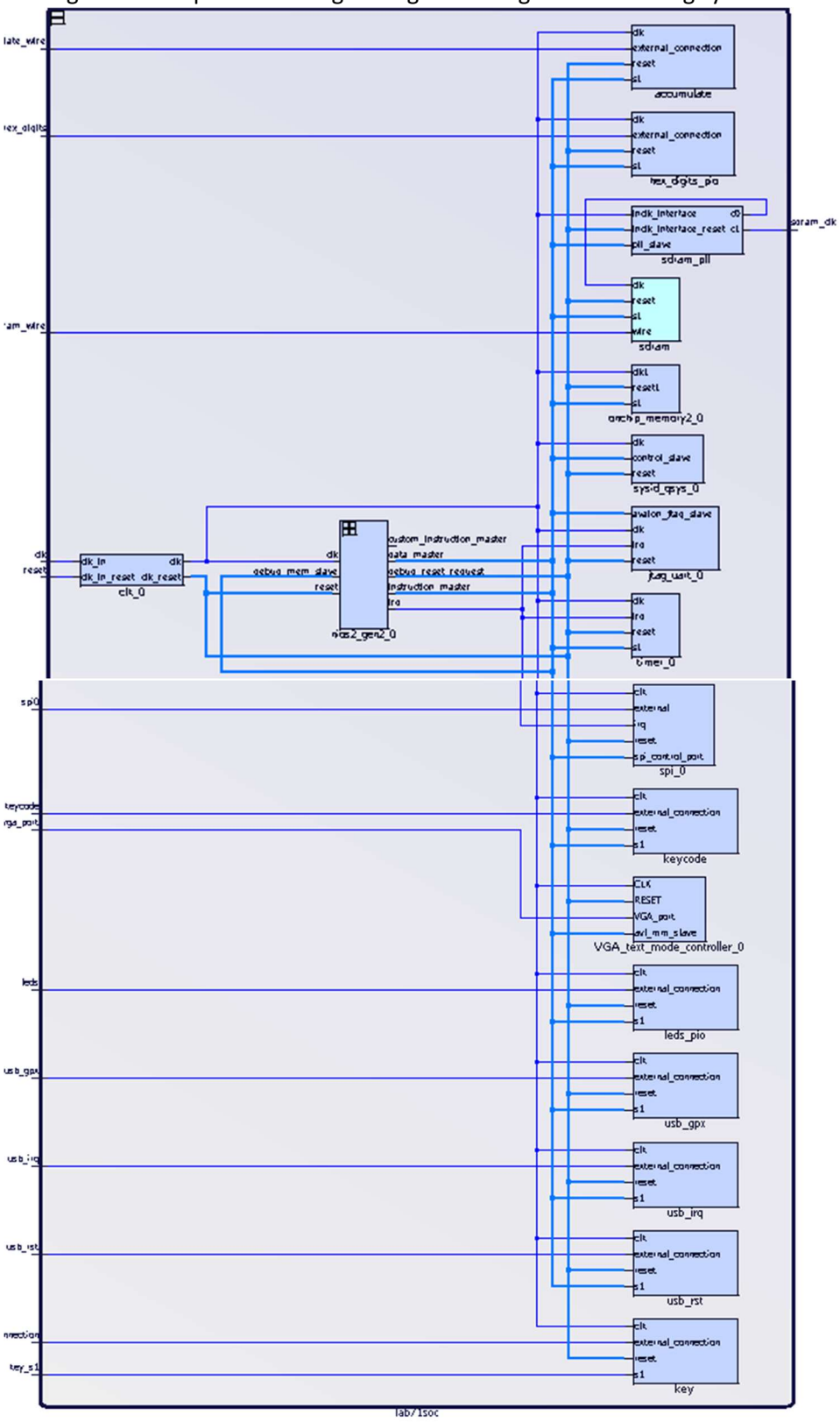
Exports: vga_port

Description: This is a ECE 385 custom IP that we created to interface with on chip memory as a more advanced color mapping module

Purpose: This custom IP is required to have the necessary control over the VGA output



Putting the entire platform designer together we get the following synthesis:



This is the entire platform designer combined together within a sythesis diagram

Design Resources and Statistics

Below are the design resources for 7.1

LUT	36,149
DSP	0
Memory(B-Ram)	0
Flip-Flop	2,200
Frequency	50 Mhz
Static Power	97.70mW
Dynamic Power	306.81mW
Total Power	426.22mW

Below are the design resources for 7.2

LUT	5,392
DSP	0
Memory(B-Ram)	0
Flip-Flop	320
Frequency	50 Mhz
Static Power	96.54mW
Dynamic Power	65.65mW
Total Power	184.22mW

Looking at both the tables toegther we can see that the resources in 7.1 are much higher. It uses over 36,000 LUTs compared to 7.2 which uses a little over 5,000. 7.2 is much more efficient in resources and also disspates far less power. The tradeoffs make 7.2 the clear option to go for in terms of efficiency and space.

Conclusion

Our design had full functionality. We were able to pass through the given sequences for 7.1 and 7.2. One of our only main bugs came in 7.1. We initially tried to get the design to work with OCM instead of registers but eventually gave up and switched to registers, as we found it too challenging to debug, as we were unsure if it was our logic or the OCM that had a bug.

This lab will be very useful when it comes to the final project. The ability to print out sprites using palettes is much easier than trying to print out a sprite manually. The interaction with the microprocessor is also very helpful as we now have a way to use C code to print things on the VGA monitor.

The main part of the lab that was not explained correctly was the changing of the C code. There seemed to be very little direction when it came to modifying this and it was very frustrating. In

terms of the things done well, we thought that the sprite generation document did a good job of explaining the algorithm.