# ECE 385
# Spring 2023

## Lab Report 7: VGA font generation in monochrome and multicolor with use of registers and OCM

**TA: Shitao Liu**
**Partners: adnanmc2 and akshat4**

## Introduction:

In this lab, we created a simplified text mode graphics controller which is connected to the Avalon memory-mapped bus and supports 80 column text mode through the VGA output. For the Lab 7.1, we created a simplified version of the monochrome graphics adapter, supporting only black and white text. For Lab 7.2, we redesigned and extended the graphics controller from week 1 around on-chip memory, enabling a much larger video memory (VRAM) space. This will allow you to draw different sets of text font in various colors as chosen by us with the use of a "color palette".

Lab 7 delved into the use of a **video graphics array (VGA)**, which is a standard for displaying graphic driver signals to a screen. In Lab 7.1, we simply used registers to write characters to a screen via a VGA connection in a monochrome format (not black and white, but just a background and foreground color). In Lab 7.2, we significantly increased the number of characters to write to the screen and implemented multicolor functionality. Now that there were too many registers for the DE10 Lite Board to support with hardware, we migrated to on-chip **memory**, using a **dual port M9K block** to read and write.

The tools and concepts covered in Lab 6 are built upon within Lab 7 through the introduction of a more detailed implementation of a VGA text interface. Initially, we were just using the NIOS-II processor to run C code that interacted with FPGA hardware so that we could control the movement of an onscreen ball via a USB keyboard. The VGA controller for lab 6 was simply used to print pixel values for a ball and a background to the screen. In Lab 7, we created a font generator by connecting using the VGA controller to print glyphs to a screen, mapping information from either registers (Lab 7.1) or an M9K block (Lab 7.2) to the screen. This functionality was enabled via the use of **Avalon memory-mapped bus**, which is a common way to connect different components in a System-on-Chip (SoC) design. It holds all the modules together in order for the VGA controller to create the desired output on the screen in real time.

## Written description of Lab 7 system:

Our introduction mentions the new tools and concepts covered in Lab 7. Here, we will explain how they were used to create the framework for our font generator. The basic overview for Lab 7.1 involved an Avalon bus module containing 601 32-bit registers. 600 of these registers held **word addresses**, with each word address/register containing sets of bits encoding 4 glyphs. Within the module, we implemented logic that would map the position of the DrawX and DrawY variables (x and y position of the pixel gun) to the VRAM layout. This VRAM layout consists of 80 x 30 glyphs, meaning that there were 20*30 = 600 registers in total. Using our DrawX and DrawY variables, we implemented a simple algorithm to access the address of the glyph to be printed before using this address to access the relevant word information from our registers. Using this information (the state of the invert bit and the CHAR1 bits), we accessed the font_rom_address module containing string maps of **0s** and **1s** representing shapes to be printed

onto the screen. The RGB values for a glyph would be printed to the screen via the VGA controller for pixels with their bits set at 1. And the “monochrome” color setting of the RGB values for display was done by register 601 - the control register, which contained sets of bits encoding the R, G and B values. This was the basic functionality of Lab 7.1

For Lab 7.2, extended the VRAM to support (per character) color text. This COULD be done by doubling the number of registers from 600 to 1200 and storing 2 glyphs per register instead of 4. Then, we could modify the algorithm used to access the word address for the font_rom module to account for 2 glyphs per register. However, 1200 registers is not a feasible number of hardware components for generation by the FPGA. There is simply not enough space. Therefore, we instantiated the dual-port M9K memory block and used our C code on the NIOS II processor to populate the memory locations. We now wanted to store 16 different colors in a color palette for color text generation. So we created 8 registers and stored the RGB bit sets for 2 colors in each. Now, while the C code is populating addresses up till x7FF, only port a of the OCM is being read to AVL_READDATA. The moment the topmost bit of the AVL_ADDRESS becomes a 1, indicating x800, a multiplexer receiving this top bit AVL_ADDRESS[11] as a select pin switches the AVL_READDATA to read from the color palette registers. When this happens, the AVL_WRITEDATA input starts writing to the color registers as well as the OCM memory locations. This allows us to fill the registers with the colors needed for our per character color support.

**Table 8. Peripheral Memory Map (Week 2, Color Mode)**

| Word Address Range | Byte Address Range | Description |
|---|---|---|
| 0x000 - 0x4AF | 0x0000 0000 - 0x0000 12BF | VRAM – 2 bytes per character, 2 characters per word. 80 column x 30 row. Data format is in raster order (one line at a time). |
| 0x4B0 - 0x7FF | 0x0000 12C0 - 0x0000 1FFF | Unused but reserved by Platform Designer |
| 0x800 - 0x807 | 0x0000 2000 - 0x0000 201F | Palette- 8 words of 2 colors each, for 16-color palette |
| 0x808 – 0xFFF | 0x0000 2020 – 0x0000 3FFF | Unused but reserved by Platform Designer |

Upon accessing the relevant glyph information from the VRAM words or registers stored on the OCM, the BG index, FG index and CHAR1 information are extracted. BG[3:1] and FG[3:1] specify the index of the color palette register we want to take the color from, and BG[1]/FG[1] tell us whether to choose from the right color or left color. This design choice seemed intuitive and was easy to implement while writing the logic for filling colors in the C code on NIOS II Eclipse. Lastly, the setting of the RGB being output from our vga_text_avl_interface (Avalon bus module) is handled in the same way as they were in Lab 7.1 except the color information now comes from the color palette registers instead of the control register.

Each glyph was stored as a 8x16 pixel block before being extracted and processed to the screen by our logic. The table below lists all the signals we had to deal with in both weeks of Lab 7.

**Table 1. Avalon-MM Slave Port Interface Signals**

| Name | Direction | Width | Description |
|---|---|---|---|
| **read** | Input | **1** | High when a read operation is to be performed. |
| **write** | Input | **1** | High when a write operation is to be performed. |
| **readdata** | Output | **32** | 32-bit data to be read. |
| **writedata** | Input | **32** | 32-bit data to be written. |
| **address** | Input | **10** | Address of the read or write operation. |
| **byteenable** | Input | **4** | 4-bit active high signal to identify which byte(s) are being written. |
| **chipselect** | Input | **1** | High during a read or write operation. |

For both labs, the AVL_READ, AVL_WRITE, AVL_CS signals are conditional checkers to ensure that the Avalon memory mapped bus module is ready to be interfaced with. For Lab 7.1, we had byte_enable logic to fill the registers with VRAM word addresses in the correct way. In Lab 7.2, our ONCHIPMEM instantiation automatically handled all these signals for the read, write, address, input and output signals of the memory.

## a. Week 1 (Monochrome Text Display)

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

-> The VGA Text mode controller IP is a reusable hardware design unit in the form of a HDL which is connected to the Avalon memory mapped bus and supports 80 column text mode through the VGA which can be used to print out texts in different colors and background colors. For the first week of the lab we focused on a monochrome display that allows just two different colors at the same time. This was available and possible through the values stored in the registers. The IP block was used to print on the 80*30 block/ screen.

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

-> For the first part of the lab we used 8 32-bit registers to store the texts and their respective colors. Our initial idea was to use the on-chip memory for the first part of the lab as well but we felt that we might face an issue with the difference of the clock cycles between the ram and the VGA which might lead to issues later on. The read and write logic of the registers was put inside an always_ff block. For both the operations we first check if the chip_select is true or not, if it's

not, the modules are not supposed to read or write. If the chip_select is on and if we can write onto the chip then depending the byte_enable value we write from the registers onto a specific bit section of the AVL_writedata, depending what character we aim to write. Similarly if the read_enable is turned on, we read from the AVL_read data coming from the avalon bus and display that data onto the VGA.

**iii. Describe the algorithm used to draw the text characters from the VRAM and font ROM (specifically, describe the equations required to generate the correct addresses to index into the VRAM as well as the font ROM)**

-> For the scope of this lab we were provided with the x and y coordinate, which was used to calculate the specific address to print specific characters onto the VGA screen. For the first part we were dealing with 600 characters which were stored in 32 bits addresses in either of the 8 registers. Our job was to find where they were located in those registers and getting those data to be transferred onto the VGA screen. First with the given x and y coordinate we divided with a certain number to get the row and column number that the number was located at. Then using that information we were able to calculate the row major index as all the pixels on the screen were stored on a 1D array. Then we divided the index by 2 to get rid of the offset. Then we access the value at that index. After we assess the value at the index we store the offset value in another variable to calculate. So, after finding the word, we had to find which 8 bits corresponded to our glyph. To figure this part out, we had to find the modulus of our byte address divided by four, as this would show us which if the character was the first, second, third or fourth in the word. According to that we set the invert bit and which character it corresponds to in our register. In order to output a row of 1s and 0s corresponding to the foreground and background colors, we utilize the font ROM module, which accepts an address. Since there are 27 characters in a font ROM, you must multiply the seven least significant bits by 16, which is the number of rows in a character. Then you must add the pixel's y coordinate modulo 16, which will tell you which row of the character you are in order to determine the address. Finally, use the output of the font module and the index of that row, which is 7 - drawX modulus 8, to determine whether the precise pixel you are at is in the foreground or background.

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

-> The inverse bit was used to take control of what color was to be printed in terms of RGB values and we were able to control that by making a variable get an inverse bit which was set depending on the offset value. The inverse bit in most cases was the 8th bit of the address. If the bit is high, you invert, otherwise you don't. Using the algorithm from above, we can figure out our pixel color in the case of no inversion. If there is no inversion, we XOR the value with the

invert bit. Accordingly, if the invert bit is high, you would set your real pixel to 0 if it was a 1 and 1 if it was a 0, and if it is low, you would leave the pixel exactly as it is. After ascertaining the bit's real value, we will draw the image by filling each pixel with color inside of an always_ff block. We must use the control register for this portion. Bits representing the RGB values of the background and foreground are stored in the control register. If we can draw, we check if the pixel is high or low, and assign the RGB values accordingly; if it's one we draw in the foreground, if it's zero we draw in the background.

## b. Week 2 (Color Text Display)

**i. Describe the hardware changes you had to make to support the use of multi-color text. At the minimum you must describe:**

1. **Modification of register-based VRAM to on-chip memory-based VRAM. How did your design share the limited on-chip memory ports?**

In our "megaram" function generation, we simply declared a dual port M9K block. This actually generates 2 M9K ports with 1 read and write port each, but for our purposes, since we do not care about resource utilization, it's not a problem. In theory, we can consider it as a single dual port OCM block.

The read_a port reads AVL addresses all the way through this address range specified in the overview. Meanwhile, the write_a ports inputs VRAM words and color information to all the relevant memory addresses on the OCM. This is how we use the first pair of ports. The read_b port reads glyph addresses that we generate using some basic math. There is no input to the write_b port as we do not need to do anything here; we are simply accessing the relevant word address for a glyph specified by the **font_rom module** while printing pixels to the screen via the RGB output of the VGA controller.

Ultimately, we were able to increase the amount of data it is possible to store on the FPGA by using the OCM block.

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

For Lab 7.2, we had to change the size of the AVL_ADDR parameter in the VGA_text_mode_controller IP to 12 bits in order to account for a doubling of the registers from 60 to 120. This gave us a maximum of 4096 possible addresses in the OCM. This allowed us to accommodate more VRAM registers/words and a color palette. The AVL_ADDR[11] bit was used to distinguish between the VRAM words and color palette registers.

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

As seen in the image below, the only modification to the original drawing algorithm involves a change of the size of the offset parameter from 2 bits to 1 bit, as there are now only 2 glyphs per register to pick from while printing to the screen. To enable this, we now save row_major_address (address to adress_b port) as an 11 bit value that is divided by 2 instead of 4 (to shift out only 1 bit from the bottom as offsetx). And now we also store FG and BG information from the VRAM word address in order to choose the relevant colors while deciding between the setting of a background or foreground color with the pixel gun.

```
always_comb
begin
   draw_x = Drawx >> 3; // number of possible glyphs you can h
   draw_y = Drawy >> 4; // depends on position of electron gun
   row_major_index = (draw_y*80 + draw_x);
   row_major_address = row_major_index >> 1; // got index, now
   // so now we are sending this address to PORT 2 in order to

   // regval = LOCAL_REG[row_major_address]; // 10 bit value
   offsetx = row_major_index[0];// row_major_index % 2; // thi

   if(offsetx == 1'b0)
   begin
      invert = READ_B[15];
      CHAR1 = READ_B[14:8];
      FG = READ_B[7:4];
      BG = READ_B[3:0];
   end


   else // else if(offsetx == 1'b1)
   begin
      invert = READ_B[31];
      CHAR1 = READ_B[30:24]; // this is the FONTROM CODE... 7
      FG = READ_B[23:20];
      BG = READ_B[19:16];
   end

   // fontRom is another row major thing... 16 pixels per glyp

   fontaddr = (CHAR1 * 16 + (Drawy[3:0]));
   // glyph changes every 16 pixels on the y axis

   out = data[7 - Drawx[2:0]]; // this data comes back from th

   // glyph changes every 8 pixels on the x axis
   // this out is a single bit
```

4. **Additional modifications necessary to support multicolored text.**

For this, we created a set of 8 color registers that store 32 bit values. This is our color palette. Each register stores the RGB values for 2 distinct colors, which we fill in through addresses reserved for the colors. The FG and BG indices were then used to choose between the left and right color while setting the Red, Green and Blue values of our VGA controller in our vga_text_avl_interface. At any given time, the pixel gun is either setting a foreground pixel's

color or a background pixel's color. Therefore, we used the FG[0] and BG[0] bits to determine whether to set the pixel of either to the first color or second color in the color palette register accessed.

```
if (blank) begin
   if (out ^ invert == 1'b0) begin
      //RGB <= color_registers[BG[3:1]][12 + (offset1*8): 1 + offset1*12]; // setting background colors
      if (BG[0] == 0)
         RGB <= color_registers[BG[3:1]][12:1];
      else
         RGB <= color_registers[BG[3:1]][24:13];
   end

   else begin
      //RGB <= color_registers[FG[3:1]][12 + (offset2*8): 1 + offset2*12]; // setting foreground colors
      if (FG[0] == 0)
         RGB <= color_registers[FG[3:1]][12:1];
      else
         RGB <= color_registers[FG[3:1]][24:13];
   end
will this cause a clock cycle of delay since always_ff blocks are sequential?
   red <= RGB[11:8];
   green <= RGB[7:4];
   blue <= RGB[3:0];
```

## 5. Additional hardware/code to draw paletted colors.

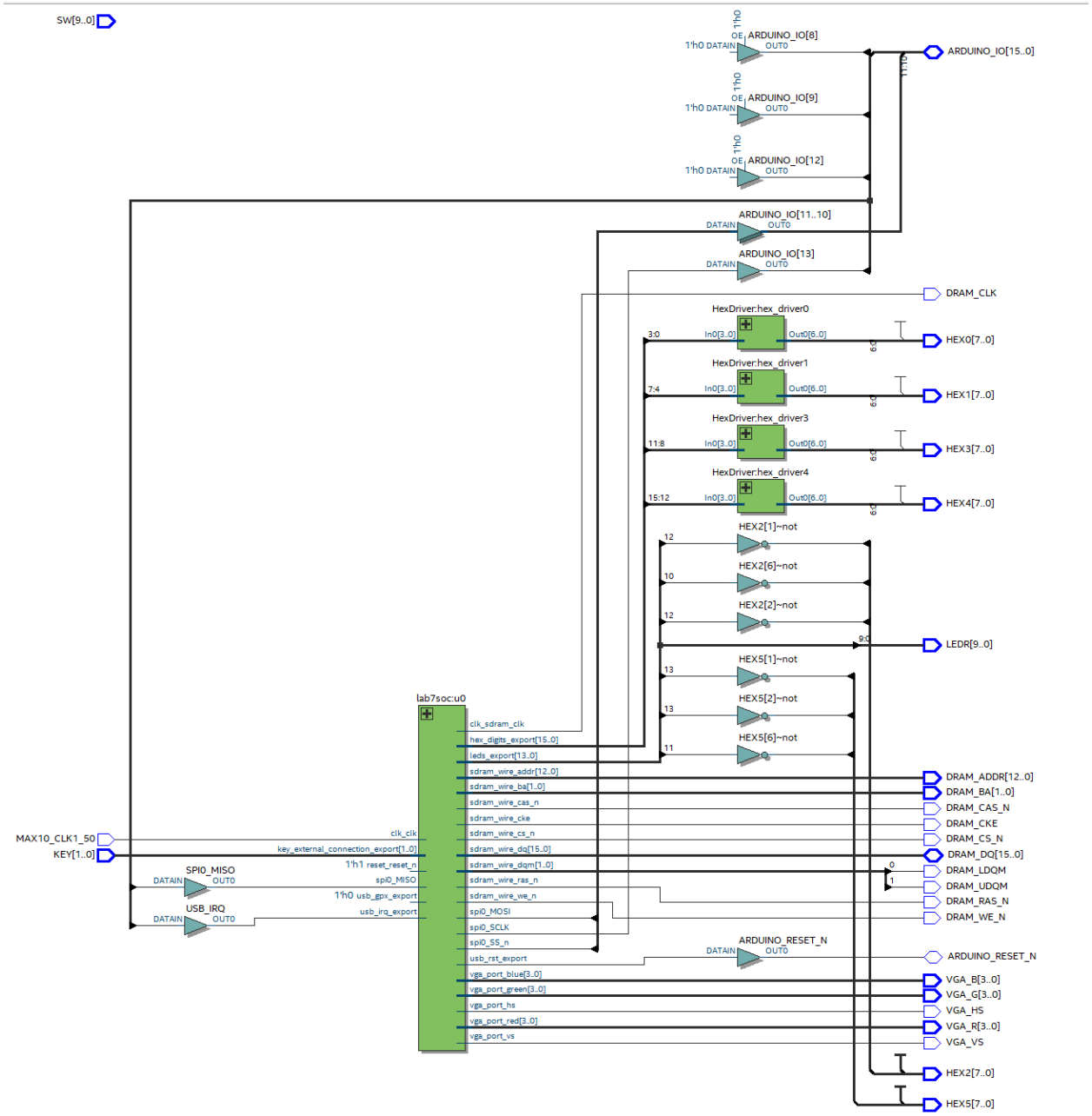
In SystemVerilog, we simply declared an unpacked array with packed dimension **32** and unpacked dimension specified by **8** to create our "color palette". The code is specified in the attached images and the prior paragraph. After that, we modified sections of our C code to fill in the relevant colors.

```
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 == 0){
        vga_ctrl->palette[color/2] &= 0xFFFFE000;
        vga_ctrl->palette[color/2] += (red << 9) + (green << 5) + (blue << 1);
    }

    else{
        vga_ctrl->palette[color/2] &= 0x00001FFF;
        vga_ctrl->palette[color/2] += (red << 21) + (green << 17) + (blue << 13);
    }

}
```

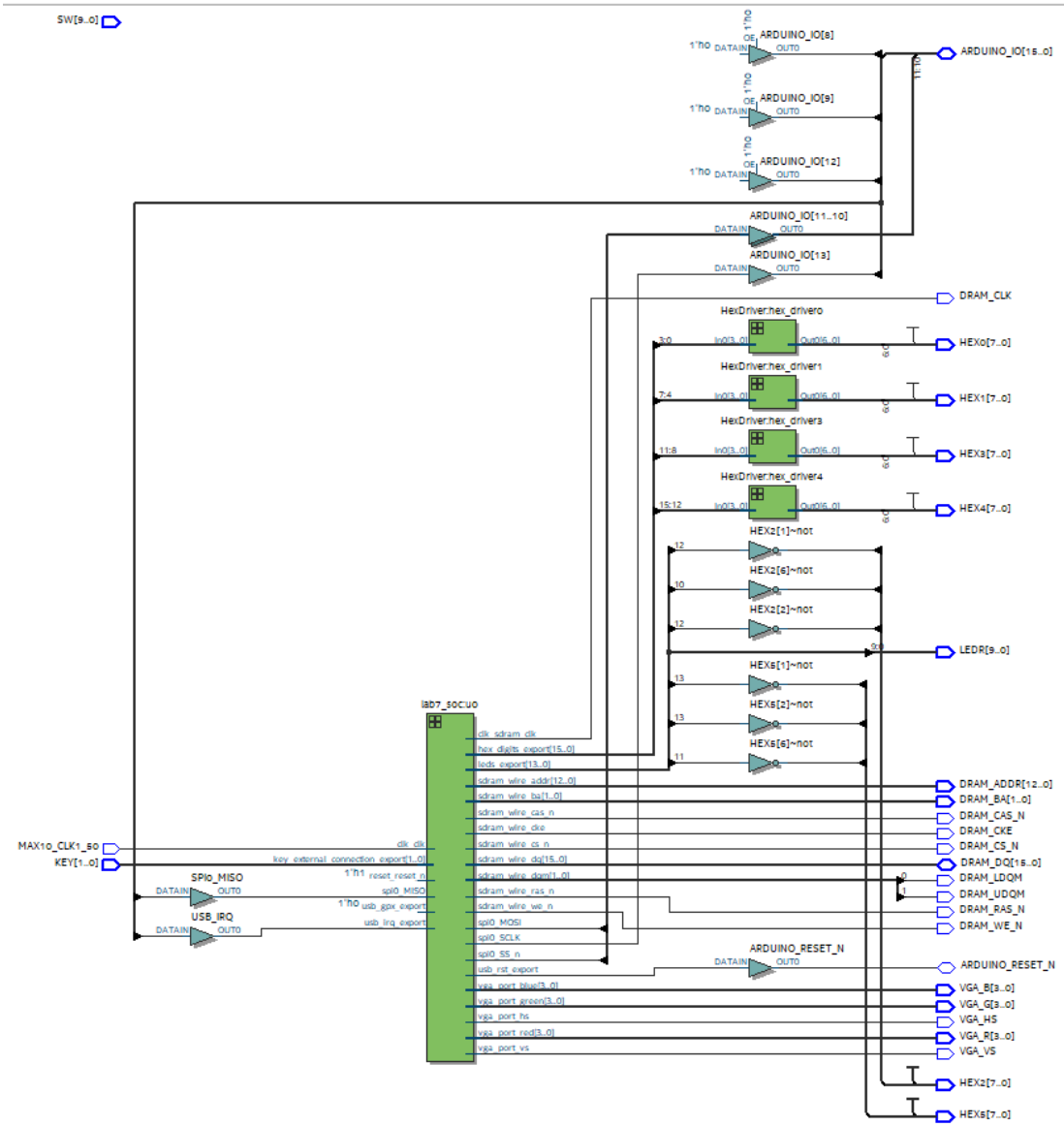
As seen here, we wrote the function **setColorPalette.** This function bits shifts the RGB values into a palette being pointed at by **vga_ctrl.**

# Block diagram:

## Lab 7.1 Top_level view:



**Lab 7.2 Top-Level View**




## Module descriptions and IP block documentation:

### 1) Module: lab7.sv

**Purpose:** The purpose of the module was to instantiate all other modules and make them interact with each other by connecting them. Furthermore, it's also the top level module

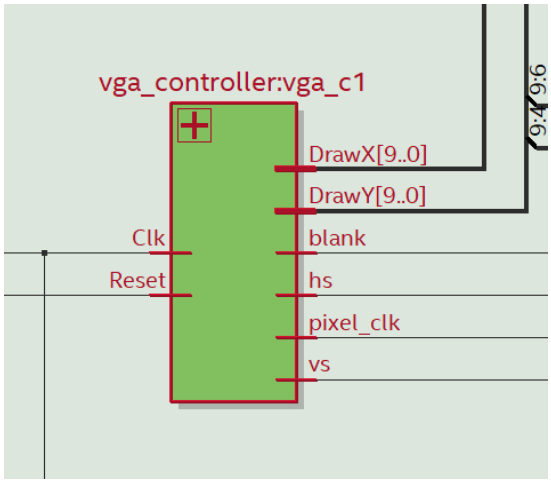
**Input and output signals:**

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

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

```
//////// SW ////////
input    [ 9: 0]   SW,
//////// LEDR ////////
output   [ 9: 0]   LEDR,
//////// HEX ////////
output   [ 7: 0]   HEX0,
output   [ 7: 0]   HEX1,
output   [ 7: 0]   HEX2,
output   [ 7: 0]   HEX3,
output   [ 7: 0]   HEX4,
output   [ 7: 0]   HEX5,
//////// SDRAM ////////
output             DRAM_CLK,
output             DRAM_CKE,
output   [12: 0]   DRAM_ADDR,
output   [ 1: 0]   DRAM_BA,
inout    [15: 0]   DRAM_DQ,
output             DRAM_LDQM,
output             DRAM_UDQM,
output             DRAM_CS_N,
output             DRAM_WE_N,
output             DRAM_CAS_N,
output             DRAM_RAS_N,
//////// VGA ////////
output             VGA_HS,
output             VGA_VS,
output   [ 3: 0]   VGA_R,
output   [ 3: 0]   VGA_G,
output   [ 3: 0]   VGA_B,
//////// ARDUINO ////////
inout    [15: 0]   ARDUINO_IO,
inout              ARDUINO_RESET_N
```

**Description:** This module was responsible for instantiating the vga_controller module, ball module, and color_mapper module. The instantiations connect the hardware components with the software and allow the keyboard peripherals to interact with the monitor peripherals.

2) **Module: VGA_controller.sv**




**Purpose:** The purpose of this SV file was to instantiate the functionality of VGA controller along with the specification of the horizontal and vertical sync

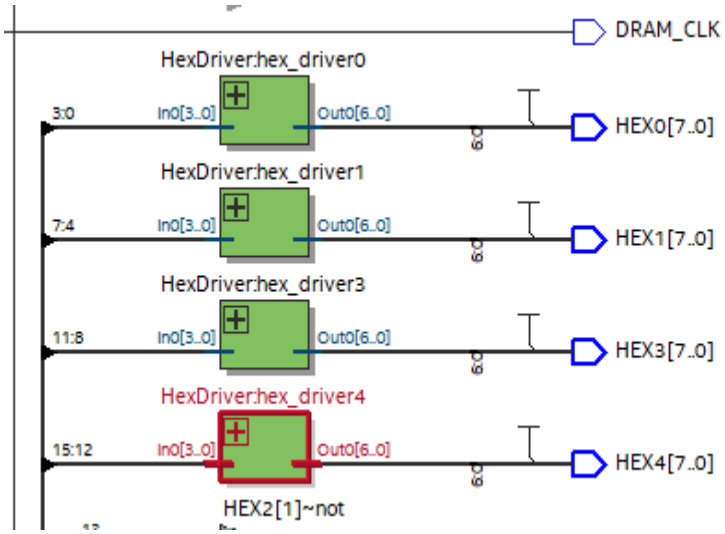
**Inputs:** CLK, Reset

**Outputs:** logic[9:0]DrawX, DrawY

logic hs, vs, blank, sync, pixel_clk

**Description:** This module is responsible for handling the horizontal and vertical sync of each screen and setting each pixel with the help of each electron beam gun behind the screen.

3) **Module: Hexdriver.sv**

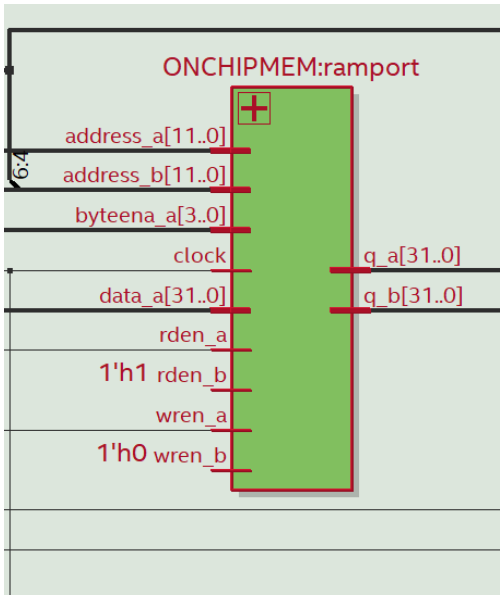



**Inputs**: [3:0] In0

**Outputs**: [6:0] Out0

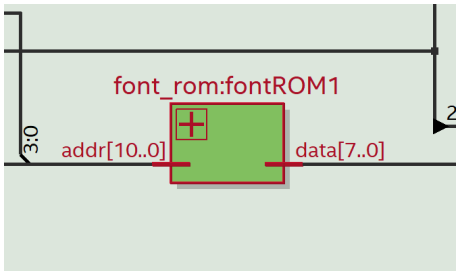
**Descriptions**: This module contains case statements to make initiations that show how each hex value in this module corresponds to the hex value which would be seen on the DE-10 board.

4) **Module**: ONCHIPMEM (file: ONCHIPMEM.v) **(LAB 7.2 ONLY)**




**Input:** [11:0] address_a, [11:0] address_b, [3:0] byteena_a, clock, [31:0] data_a, [31:0] data_b, rden_a, rden_b, wren_a, wren_b
**Output:** [31:0] q_a, [31:0] q_b
**Purpose:** The ONCHIPMEM module instantiates an M9K memory block.
**Description**: This module is made up of two separate memory banks, A and B, each with a separate address input, read enable, write enable, byte enable, and data input/output ports. The module synchronizes the read and write operations using a shared clock signal. The module produces the data saved at the desired address when a read operation is requested. The module writes the requested data to the requested address when a write operation is requested. The read and write enable signals regulate the read and write operations, respectively, whereas the byte enable signals permit single-byte writes to memory.

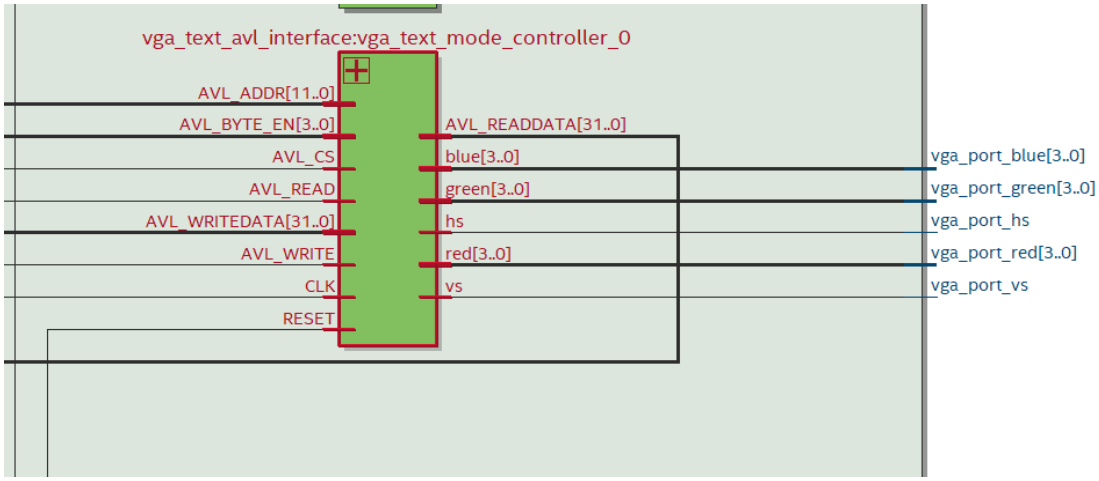
5) **Module**: font_rom (file: font_rom.sv)




**Input:** [10:0] addr
**Output:** [7:0] data
**Purpose:** Implements asynchronous ROM for writing glyph data

**Description:** We don't make any changes into this file but we take the individual x and y values of the specific character and display that on to the screen. It stores the values in the form of 0s and 1's. The font_rom module implements an asynchronous ROM that contains 8-bit rows of pixel data for each glyph used in the VGA text mode.

6) **Module**: vga_text_avl_interface (file: vga_text_avl_interface.sv)



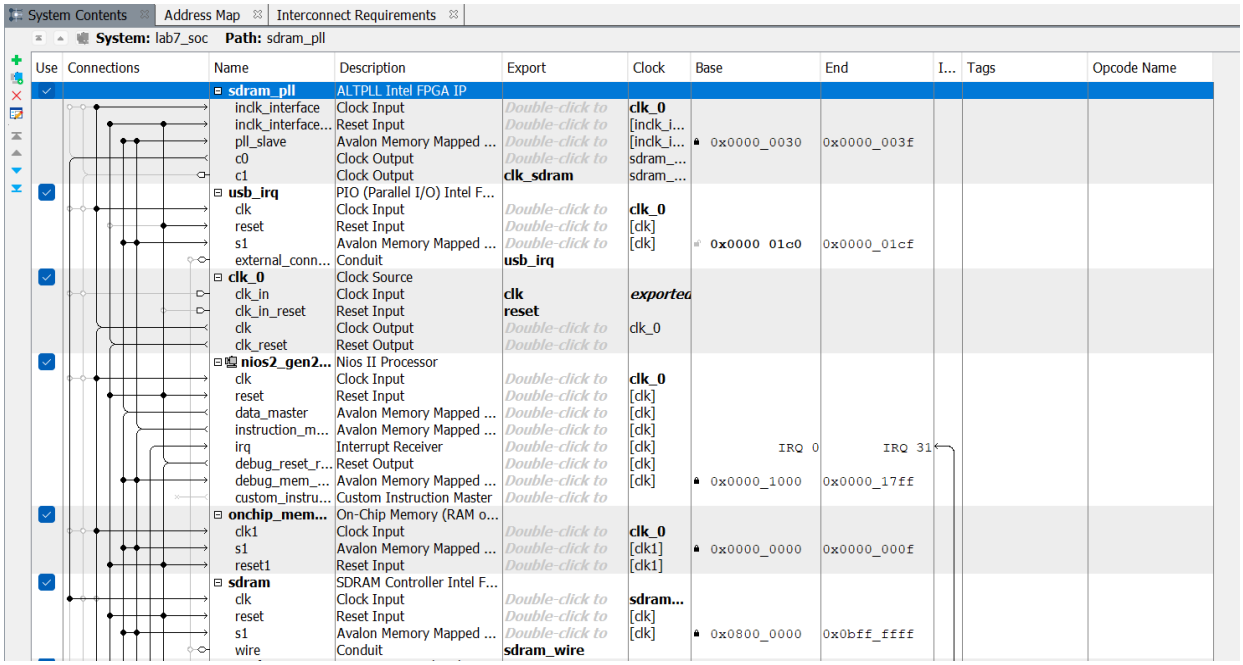

**Input:** CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [11:0] AVL_ADDR, [31:0] AVL_WRITEDATA

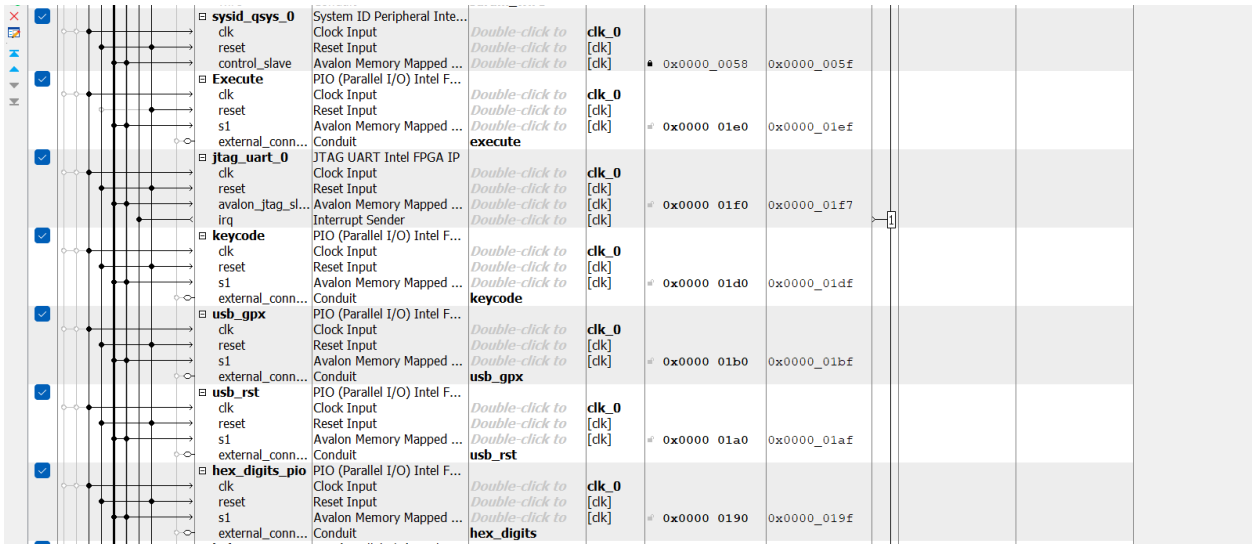
**Output: [**3:0] (red, green, blue), hs, vs

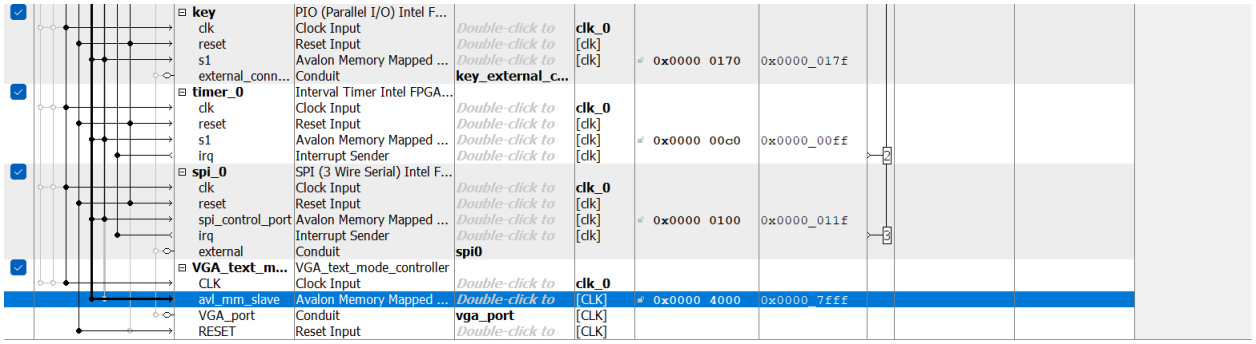
**Purpose**: To handle logic for printing fonts in both labs and act as our **avalon memory mapped bus** with all the avalon (AVL signals)

**Description:** This module contains all our logic and does all the processing needed to produce RGB and hs/vs information for the vga ports to print fonts to the screen in both Lab 7.1 and Lab 7.2. This module's functions have been covered in detail throughout the report, and it instantiates ONCHIPMEM, vga_conntroller and ram_port in order to tie everything together. For Lab 7.1, this module stores 601 registers for monochrome font outputs while in Lab 7.2 it instantiates the ONCHIPMEM and handles all the relevant logic for storing and using the color palette registers.

# Platform Designer:






1) **LED:** The LED was defined as a PIO(Parallel I/O) with specifications as an output. The LED was used to display the output from the accumulator which was calculated by taking in the inputs from the switches.

2) **Switches:** Similar to the previous component the switches were defined as PIO(Parallel I/O ) but they were used to take data from the user by flipping the switches that are present on the DE-10 Lite FPGA board.

3) **Clk_0:** This is a 50 MHz clock from the FPGA that is used to synchronize a clock and a reset which are used to clock and reset other modules.

4) **Nios2_gen2_0:** This module is used to instantiate the module that is used to define the NIOS II processor that we use to perform various tasks like writing C/C++ on eclipse to making these modules on the platform designer.This makes it easy for software developers to write and debug code for the processor, reducing the time and effort required for development. One of the key advantages of the NIOS II processor is its ease of use.

5) **SDram:** SDRAM is the memory control module which is responsible for providing an interface between the processor and SDRAM itself. It used to reserve memory blocks, initialize data and provide functionality for initial read and write functions as well. When the Nios II processor needs to access data from the SDRAM, it sends a read request to the memory controller. The memory controller retrieves the requested data from the SDRAM and sends it back to the processor. Similarly, when the processor needs to write data to the SDRAM, it sends a write request to the memory controller, which writes the data to the appropriate memory location. The SDRAM's speed and capacity can affect the overall performance of a Nios II system.

6) **Sdram_pll:** The SDRAM clock is linked to a clock signal that is intentionally delayed by 1 ns. This delay is implemented to compensate for any phase discrepancies that may arise between the master and slave clocks.

7) **Sysid_qsys_0:** This is used for checking errors in the connections from the hardware to software components that are connected.

8) **Jtag_uart_0:** This module was instantiated in the platform designer to allow for easy debugging of the C code on the Eclipse by allowing us to communicate with the Command line terminal and commands. The JTAG UART is a hardware device which allows for communication between the FPGA and other devices. The JTAG UART module allows for bi-directional communication between the device and an external device or computer

9) **Keycode:** The keycode module is used for instantiating the keyboard hardware device which is used as a communication input device into the FPGA. It enables the processor to retrieve unique codes that correspond to a pressed key on a USB keyboard.
10) **usb_irq:** This module is required to for the interaction between the keyboard and the FPGA as it sends an interrupt signal which is used by the USB device to notify the host (computer) about an event that requires attention, such as the insertion of a new device or the completion of a data transfer.
11) **usb_gpx:** This is another module which is required for communication between the FPGA and the keyboard, GP Stands for USB General Purpose Input/Output. It provides a set of pins that can be configured by the device designer for various purposes, such as controlling LEDs, reading switches, or interfacing with sensors.
12) **Usb_rst**: A signal used to reset the USB device or put it in a low-power mode. It is typically activated by the host, either through a software command or a hardware pin. This is needed to utilize the USB keyboard with the FPGA.

13) **HEX_digits_pio:** This module is used for instantiating the Hex displays on the FPGA which are responsible for displaying the ASCII value for each character that is pressed on the keyboard. By utilizing a 16-bit parallel interface, the transfer of multiple digits can be accomplished simultaneously, resulting in faster and more efficient communication.

14) **Timer_0**: Used to send signals to the FPGA so that NIOS II can check time passed.

15) **Spi_0**: This module is instantiated to allow the devices to communicate over the API protocol. More specifically it allows the slave and master channels to communicate over MISO and MOSI datapaths

16) **VGA_text_mode_controller**: This module is defined to take care of displaying multiple characters and colors on to the screen. It is a programmable PIO block which allows communication with the VGA screen. This block consists of two primary components, namely the input/output signals and the Avalon-MM slave port. The Avalon -MM slave port is responsible for read/write/chip select and various other operations associated with the Avalon bus.

**Document the Design Resources and Statistics from the lab manual. Each week's design should have different design statistics, and you should briefly discuss the difference between using on-chip memory for VRAM and registers. Which design is more efficient, what are the tradeoffs?**

**Lab 7.1 (Week 1) Design Resources and Statistics**

| **Parameter** | **Value** |
|---|---|
| LUT | 36122 |
| DSP | 0 |
| Memory (BRAM) | 13671 |
| Flip-flop | 22380 |
| Frequency | 68.04 MHz |
| Static power | 101.12 mW |
| Dynamic power | 283.36 mW |
| Total power | 355.57 mW |

**Lab 7.2 (Week 2) Design Resources and Statistics**

| **Parameter** | **Value** |
|---|---|
| LUT | 5324 |
| DSP | 0 |
| Memory (BRAM) | 142464 |
| Flip-flop | 2856 |
| Frequency | 70.14 MHz |
| Static power | 96.18 mW |
| Dynamic power | 0.69 mW |
| Total power | 106.18 mW |

The storage capacity and access time of on-chip memory versus registers for VRAM are the primary distinctions. Compared to registers, on-chip memory can store a lot more data, but it is slower to access. We used 600 registers in Lab 7's first week to store the monochrome graphics controller's VRAM words (glyphs). We could store four glyphs in each register because they can each hold 32 bits of data. Due to faster access times for registers than for on-chip memory, this approach is faster overall. Additionally, employing registers does not need allocating address space for the memory, which in some architectures can be advantageous.

6) **Conclusion**

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

-> We were able to get the whole lab done within the given time frame. Initially we didn't entirely understand the core of the lab but with the lab of the CA's and the professor we were able to understand and complete the labs for both the weeks.

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

**->** As we are aiming to build a video game for our final project, we will be dealing a lot with manipulation of pixel colors and graphics. This lab gave us a good overview of how to deal and manipulate the pixels on the screen to display objects according to our needs. Furthermore we learnt on how to reach with the font_rom.sv to display objects and colors onto the VGA screen. A few extensions to the lab 7 designs are: Video games.

**c. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.**

**->** The lab was in general a great way to learn more about pixels and their interactions with the VGA screen but the documentation for this lab was very limited and not descriptive enough. Most of the students had to sit down in office hours to have a good grasp on how to set the RGB values and how they interact with the font rom to get the required colors and