*University of Illinois at Urbana-Champaign*

# ECE 385

**LAB 7**

LAB REPORT

by Shubham Gupta & Devul Nahar (sg49 & danahar2)

3/04/2022

## Introduction:

The purpose of this lab is to draw glyphs on the VGA monitor with foreground and background colors. In the first week, we display glyphs on the screen using two main colors: foreground and background. We also have the ability to invert the color (to give the effect of highlighting text). In this week we learned about how the VGA monitor works as well as gained a deeper insight into how to interact between register-based memory, nios, and the underlying complexities of integrating hardware and software. For this week we stored all the glyphs on register-based memory (601 in total – 600 registers for glyphs and 1 control register storing foreground/background information). For the second week, we extend our week 1's design so that each glyph can now have one of 16 colors. This means that we needed a lot more space to store the glyph information. This is why we resorted to use the SDRAM instead. In addition, we used a color palette to store the 16 colors.

In lab 6.2, we display a ball on the VGA screen. In doing so we learned a lot about SPI protocols and VGA interface. This lab extends this design, so we naturally use a lot of the stuff we had previously set up in lab 6.2 Platform Designer. Lab 6 also gave us an introduction NIOS, which we continued using for lab 7.

## Written Description and Diagrams of NIOS-II System:

### Week 1:

1. **Written Description of lab 7.1:**

   In lab 7.1 we make a monochrome display. We have a total 128 glyphs which are unique, and are given to us which are stored in font ROM. We can display 80 x 30

characters on the screen where each character is 16 x 8 pixels. We are also given a foreground and a background color for the entire screen. Based on the invert bit we draw the characters with either foreground color or background color (when invert is high).

Now to implement this we created a new IP called VGA interface. This module basically consists of a VRAM made up of 601 registers which are 32-bit. We have 600 registers for the VRAM and 1 as the control register. Since we have 600 registers, in the Avalon bus we require 10 bits to uniquely address each register. Since 2^10 is 1024.

Since we are implementing the VRAM via registers from scratch, we assume that read wait is 1 and write wait is 0. Since there are 128 glyphs, we need 7 bits to uniquely represent each glyph. We need 1 bit to represent the invert bit. Therefore, to represent the glyph and the inverted bit, we need a total of 8 bits. Each register is 32 bits and therefore each can store 32/8 = 4 characters' information.

| Bit | 31 | 30-24 | 23 | 22-16 | 15 | 14-8 | 7 | 6-0 |
|---|---|---|---|---|---|---|---|---|
| Function | IV3 | CODE3 | IV2 | CODE2 | IV1 | CODE1 | IV0 | CODE0 |

IVn = Inverse bit N
CODEn = Glyph code from IBM Codepage 437

Therefore, we need a total of 80 x 30 bytes, that is 2400 bytes of data. The Avalon bus can access all these 2400 bytes to change the VRAM as desired. The VGA interface used DrawX and drawY coordinates to decide the RGB base of the invert bit and the control register.

**2. Describe your high-level VGA Text Mode Controller IP:**

We created the IP to read and write to the VRAM via the memory mapped salve Avalon bus. The IP mainly provides the read and write functionality for the NIOS II.

We first check which 32-bit register might have the character data stored based on X and Y coordinates of the screen. Then we decide which 8-bit slice is the one that has the character information stored.

Upon getting the right 8-bit slice of the 32-bit register among the 600 registers, we go to the address of the glyph provided by the 7-bits of the 8-bit slice.

Then we decide which row of the glyph ROM we are required to get data from, again based on X and Y coordinates. Once we have the right row from the font ROM, we check for the invert bit.

Based on the invert bit, we decide the RGB values to be outputted. If the invert bit is 0, then we output the foreground RGB if the glyph row has a 1, else we output the background RGB and vice versa. These RGB values for background and foreground are stored on the control register which is 32-bits as follows:

| Bit | 31-25 | 24-21 | 20-17 | 16-13 | 12-9 | 8-5 | 4-1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Function | UNUSED | FGD_R | FGD_G | FGD_B | BKG_R | BKG_G | BKG_B | UNUSED |

BKG_R/G/B = Background color, flipped with foreground when IVn bit is set
FGD_R/G/B = Foreground color, flipped with background when IVn bit is set

**3. Describe logic used to read and write VGA registers:**

The Avalon MM bus is the primary mode of communication with our IP. Therefore, to read/write data to the VRAM, the data would have to be sent to the VGA interface via the Avalon MM bus. Our bus has the data in form of AVL_READ, AVL_WRITE, AVL_CS, AVL_READDATA, AVL_BYTE_EN, and AVL_WRITEDATA. Since we might want to

write to specific location on the screen, we need to have a Byte Enable. The description of Byte Enable is as follows:

| byteenable[3:0] | Write Action |
|---|---|
| 1111 | Write full 32-bits. |
| 1100 | Write the two upper bytes. |
| 0011 | Write the two lower bytes. |
| 1000 | Write byte 3 only. |
| 0100 | Write byte 2 only. |
| 0010 | Write byte 1 only. |
| 0001 | Write byte 0 only. |

Since we need to read/write synchronously, we need to have our code in always_ff block that is executed at every rising edge of the clock cycle. First, we see that if AVL_CS (chip select) is high or not. If it is high, we need to know if we are reading or writing. To determine that we look at the AVL_WRITE, if it is high, we know that we need to write. Else we check if AVL_READ is high, in that case we read.

For the WRITE case, we look at which segment of the 32-bit we need to write the data into. This is implemented by the case statement. In every case we simply write to the VRAM at the AVL_ADDR provided.

For the READ case, we simply set the data as AVL_ADDR in VRAM to AVL_READDATA as output. This is the explanation of our read/write.

```
52  // Read and write from AVL interface to register block, note that READ waitstate = 1, so this should be in always_ff
53  always_ff @(posedge CLK) begin
54      if(AVL_CS) begin
55          if(AVL_WRITE) begin
56              case (AVL_BYTE_EN)
57                  4'b1111 : LOCAL_REG[AVL_ADDR] <= AVL_WRITEDATA;
58                  4'b1100 : LOCAL_REG[AVL_ADDR][31:16] <= AVL_WRITEDATA[31:16];
59                  4'b0011 : LOCAL_REG[AVL_ADDR][15:0] <= AVL_WRITEDATA[15:0];
60                  4'b1000 : LOCAL_REG[AVL_ADDR][31:24] <= AVL_WRITEDATA[31:24];
61                  4'b0100 : LOCAL_REG[AVL_ADDR][23:16] <= AVL_WRITEDATA[23:16];
62                  4'b0010 : LOCAL_REG[AVL_ADDR][15:8] <= AVL_WRITEDATA[15:8];
63                  4'b0001 : LOCAL_REG[AVL_ADDR][7:0] <= AVL_WRITEDATA[7:0];
64              endcase
65          end
66          else if(AVL_READ) begin
67              AVL_READDATA <= LOCAL_REG[AVL_ADDR];
68          end
69      end
70  end
```

**4. Describe the algorithm used to draw the text characters from the VRAM and font ROM:**

```
85  font_rom myROM (.addr(gylf_final_addr), .data(gylf_data));
86
87  always_comb begin
88      temp = (drawysig[9:4] * 80) + drawxsig[9:3]; // = ((Y/16) * 80) + (X/8)
89      gylf_row = drawysig[3:0]; // = Y % 16
90      cur_col = drawxsig[2:0];  // = X % 8
91      base_addr_of_reg_VRAM = temp[11:2]; // = temp / 4
92      char_num = temp[1:0]; // = temp % 4
93
94      gylf_final_addr = {gylf_base_addr, gylf_row};
95
96      case(char_num)
97          2'b00 : begin
98              gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][6:0];
99              invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][7];
100         end
101
102         2'b01 : begin
103             gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][14:8];
104             invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][15];
105         end
106
107         2'b10 : begin
108             gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][22:16];
109             invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][23];
110         end
111
112         2'b11 : begin
113             gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][30:24];
114             invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][31];
115         end
116     endcase
117 end
```

To be able to draw the characters from the VRAM and font ROM, we essentially just need the row data of 8 bits from the ROM file and the invert bit. Based on this we can simply generate our RGB output signals. But to get these, we have used the following algorithm:

We first check which 32-bit register might have the character data stored based on X and Y coordinates of the screen. Then we decide which 8-bit slice is the one that has the character information stored.

To do this we use a variable temp which gives us the row and column of the character based on the X and Y coordinate.

```
88      temp = (drawysig[9:4] * 80) + drawxsig[9:3]; // = ((Y/16) * 80) + (X/8)
```

Then we decide which 32-bit register we need to look into to find the character data:

```
91      base_addr_of_reg_VRAM = temp[11:2]; // = temp / 4
```

Then we decide which 8-bit slice is the one that has the character information stored. To do this:

```
92      char_num = temp[1:0]; // = temp % 4
```

Upon getting the right 8-bit slice of the 32-bit register among the 600 registers, we get the address of the glyph provided by the 7-bits of the 8-bit slice and the invert bit.

```
96      case(char_num)
97          2'b00 : begin
98              gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][6:0];
99              invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][7];
100         end
101
102         2'b01 : begin
103             gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][14:8];
104             invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][15];
105         end
106
107         2'b10 : begin
108             gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][22:16];
109             invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][23];
110         end
111
112         2'b11 : begin
113             gylf_base_addr = LOCAL_REG[base_addr_of_reg_VRAM][30:24];
114             invert_bit = LOCAL_REG[base_addr_of_reg_VRAM][31];
115         end
```

Then we get data from the row of the glyph ROM by:

```
85    font_rom myROM (.addr(gylf_final_addr), .data(gylf_data));
```

Once we have the right row from the font ROM, we check for the invert bit. Based on the invert bit, we decide the RGB values to be outputted. If the invert bit is 0, then we output the foreground RGB if the glyph row has a 1, else we output the background RGB and vice versa. We have used XOR to make our design efficient with the implementation of the conditions.

```
119   always_ff @ (posedge VGA_Clk) begin
120       if (blank == 1) begin
121           if((invert_bit ^ gylf_data[7 - cur_col]) == 0) begin
122                   red = background[11:8];
123                   green = background[7:4];
124                   blue = background[3:0];
125           end else begin
126                   red = foreground[11:8];
127                   green = foreground[7:4];
128                   blue = foreground[3:0];
129           end
130       end else begin
131           red = 0;
132           green = 0;
133           blue = 0;
134       end
135   end
```

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

Our implementation of the control register is very simple. It is a 32-bit register initialized in the VGA interface module along with the VRAM. It is the 601st register in the VRAM array and can be updated via Avalon MM Bus. The data is stored in it as following:

| Bit | 31-25 | 24-21 | 20-17 | 16-13 | 12-9 | 8-5 | 4-1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Function | UNUSED | FGD_R | FGD_G | FGD_B | BKG_R | BKG_G | BKG_B | UNUSED |

BKG_R/G/B = Background color, flipped with foreground when IVn bit is set
FGD_R/G/B = Foreground color, flipped with background when IVn bit is set

**Our implementation of the invert bit is described as following:**

Based on the invert bit, we decide the RGB values to be outputted. If the invert bit is 0, then we output the foreground RGB if the glyph row has a 1, else we output the background RGB and vice versa.

This is implemented efficiently as an XOR logic between invert_bit and gylf_data at 7 – #current columns based on X coordinate:

```
always_ff @ (posedge VGA_Clk) begin
    if (blank == 1) begin
        if((invert_bit ^ glyf_data[7 - cur_col]) == 0) begin
                red = background[11:8];
                green = background[7:4];
                blue = background[3:0];
        end else begin
                red = foreground[11:8];
                green = foreground[7:4];
                blue = foreground[3:0];
        end
    end else begin
        red = 0;
        green = 0;
        blue = 0;
    end
end
```

We do glyf_data[7 – cur_col] because the bits are laid right to left but we want to print from left to right. Since a column is 8 bits, index 7 is the max. Hence, we subtract from 7 to correct the **lateral inversion**.

Apart from that, we check the **active low** blanking interval signal. As per the working of the VGA display, we are supposed to output black if we are in the blanking interval, else we are supposed to output the desired data. This is implemented in the above always_ff block using an if condition. This concludes our section for lab 7.1.

Week 2:

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

   Note that in week 1 implementation, data was stored in 600 registers, where each register was 4 bytes. However, in week 2 we wanted each character to have its own possible color. To do that each register needed to store the character as well as it its corresponding background and foreground colors. To do this we would have had to double the number of registers we use. However, since we don't have enough registers

on the FPGA, we decided to use memory-based VRAM instead. Note that one key difference between memory-based VRAM and register-based VRAM is that, register always have an input and output port, meaning that data can be accessed at any time from any of the registers. In contrast memory only has a limited number of ports. We used memory-based VRAM by instantiating a dual port RAM IP core. We used a dual port design because both the VGA controller as well as the AVALON Bus are accessing the memory at the same time. This means it would be easier for the SDRAM to have a dual port design because a user could access two different addresses of memory and read/write the same time. Once we initialized the memory with the right specifications, we instantiated the VRAM. The additional address are filled with blank space. Note that VRAM is instantiated with the following parameters:

```
VRAM myVRAM (
    .address_a(AVL_ADDR),
    .address_b(base_addr_of_reg_VRAM),
    .byteena_a(AVL_BYTE_EN),
    .clock(CLK),
    .data_a(AVL_WRITEDATA),
    .data_b(),
    .wren_a(AVL_WRITE & AVL_CS & ~AVL_ADDR[11]),
    .wren_b(1'b0),
    .q_a(temp_read),
    .q_b(Read_Data_For_VGA)
);
```

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

The only change we made in platform designer was to change the addressability from 10 to 12 bits to accommodate for the VRAM and the Palette. Note that the addressability for this design has to change from 10 bits to 12 bits. The other change that we had to do in week 2 was to extend our control register to an entire color palette of 16 colors. Note that even though we only need 11 bits to address the 1200 registers and the color palette, we use 12 bits to make it easier to determine whether we want to access

VRAM or Palette. For instance, this is easily done by looking at the 11th bit of the address: 0 for VRAM and 1 for Palette. The address divisions can be found here:

| 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 |

Additionally, we add a delay in writing to memory since OCM is synchronous as compared to registers which are asynchronous.

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

```
always_comb begin
    temp = (drawysig[9:4] * 80) + drawxsig[9:3]; // = ((Y/16) * 80) + (X/8)
    gylf_row = drawysig[3:0]; // = Y % 16
    cur_col = drawxsig[2:0];  // = X % 8
    base_addr_of_reg_VRAM = temp[11:1]; // = temp / 2
    char_num = temp[0]; // = temp % 2

    gylf_final_addr = {gylf_base_addr, gylf_row};

    case(char_num)
        1'b0 : begin
            background_idx = Read_Data_For_VGA[3:0];
            foreground_idx = Read_Data_For_VGA[7:4];
            gylf_base_addr = Read_Data_For_VGA[14:8];
            invert_bit = Read_Data_For_VGA[15];
        end

        1'b1 : begin
            background_idx = Read_Data_For_VGA[19:16];
            foreground_idx = Read_Data_For_VGA[23:20];
            gylf_base_addr = Read_Data_For_VGA[30:24];
            invert_bit = Read_Data_For_VGA[31];
        end
    endcase
end
```

Similar to week 1, the character to be printed is given by the X and Y coordinates. Essentially, we want to convert from 2D coordinates to 1D coordinates. Note that since each glyph is 16 rows by 16 columns, the address of the character is given by:

$$temp \rightarrow \frac{Y}{16} * 80 + \frac{X}{8}$$

Now, since each memory address contains only 2 glyphs, the specific memory cell address in which the glyph is stored in can be given by:

$$memory\ cell\ address \rightarrow \frac{temp}{2}$$

Note that in each memory cells there are two glyphs stored, so to decide which glyph we want to access, we do:

$$gylf \rightarrow temp\ \%\ 2$$

We get the specific row and column of the glyph we are accessing by:

$$row \rightarrow\ Y\ \%\ 16$$

$$col \rightarrow X\ \%\ 8$$

Now that we have the exact glyph data we need, we can segment the 16 bits of data according to this data:

| Bit | 31 | 30-24 | 23-20 | 19-16 | 15 | 14-8 | 7-4 | 3-0 |
|---|---|---|---|---|---|---|---|---|
| Function | IV1 | CODE1 | FGD_IDX1 | BKG_IDX1 | IV0 | CODE0 | FGD_IDX0 | BKG_IDX0 |

The final glyph address is composed of two pieces of information: which of the 128 glyphs we want to print and the row of the glyph we want to print. Once we have this address, the relevant information is accessed from font rom and then printed accordingly.

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

The first step to support multicolored text is to implement functionality that reads/writes into the 8 32-bit color palette registers. The first three bits of the address determine which of the 8 registers we need to access. Note that if the 11th bit is 1 then we are accessing the color palette and if it is 0 then we are accessing the VRAM.

```
always_ff @(posedge CLK) begin
   if(AVL_CS) begin
      if(AVL_WRITE & AVL_ADDR[11]) begin
         myColors[AVL_ADDR[2:0]] <= AVL_WRITEDATA;
      end else if(AVL_READ & AVL_ADDR[11]) begin
         AVL_READDATA <= myColors[AVL_ADDR[2:0]];
      end else if(AVL_READ & AVL_ADDR[11] == 0) begin
         AVL_READDATA <= temp_read;
      end
   end
end
```

Next, we have to set the RGB values based on the color index as well as whether we are referring to the foreground or the background. Since we have a total of 16 colors, we use 4 bits to access the colors. Since, there are only 8 palettes with two colors each (4 bits for red, green, and blue respectively), the top 3 bits determine the palette being used, and the first bit determines which of the two colors inside the palette we are referring to. The color palette looks like this:

| Address | 31-25 | 24-21 | 20-17 | 16-13 | 12-9 | 8-5 | 4-1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x800 | UNUSED | C1_R | C1_G | C1_B | C0_R | C0_G | C0_B | UNUSED |
| 0x801 | UNUSED | C3_R | C3_G_ | C3_B | C2_R | C2_G | C2_B | UNUSED |
| … | … | … | … | … | … | … | … | … |
| 0x807 | UNUSED | C15_R | C15_G | C15_B | C14_R | C14_G | C14_B | UNUSED |

Refer to the code below to see how we use the index along with foreground and background information to get the relevant RGB values:

```
always_comb begin
    color_row_b = background_idx[3:1];
    color_row_f = foreground_idx[3:1];
    color_col_b = background_idx[0];
    color_col_f = foreground_idx[0];
end

logic [11:0] background_color, foreground_color;

always_comb begin
    case(color_col_b)
        1'b0: background_color = myColors[color_row_b][12:1];
        1'b1: background_color = myColors[color_row_b][24:13];
    endcase

    case(color_col_f)
        1'b0: foreground_color = myColors[color_row_f][12:1];
        1'b1: foreground_color = myColors[color_row_f][24:13];
    endcase
end
```

```
always_ff @ (posedge VGA_Clk) begin
    if (blank == 1) begin
        if((invert_bit ^ gylf_data[7 - cur_col]) == 0) begin
            red = background_color[11:8];
            green = background_color[7:4];
            blue = background_color[3:0];
        end else begin
            red = foreground_color[11:8];
            green = foreground_color[7:4];
            blue = foreground_color[3:0];
        end
    end else begin
        red = 0;
        green = 0;
        blue = 0;
    end
end
```

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

The first thing we did in order to draw the color palette is create a struct that occupies the VRAM space in accordance with the memory division table shown above.

```
struct TEXT_VGA_STRUCT {
    alt_u8 VRAM [ROWS*COLUMNS*2]; //Week 2 - extended VRAM
    alt_u8 unused_1 [3391];
    alt_u32 myColors [8]; // same as x2000 - x201F
};
```

This function is responsible for setting the color palette to a certain RGB values. Since there are 16 colors in the color palette, we access the relevant color inside the 8 32-bit color palette, by using row = color / 2 and col = color % 2. Note that to go from a 12 bit RBG value to a 32 bit value, we need bit masking. If we are setting the first color in the color palette, then we set bits 1 to 9, and save whatever information is in bits 13 to 21. Also note that we set the relevant RGB value for certain color by shifting Red by 9 bits, then shifting Blue by 5 bits, and finally shifting green by 1 bit. We put all of this information in 32 bits by Oring the RBG values with the relevant color.

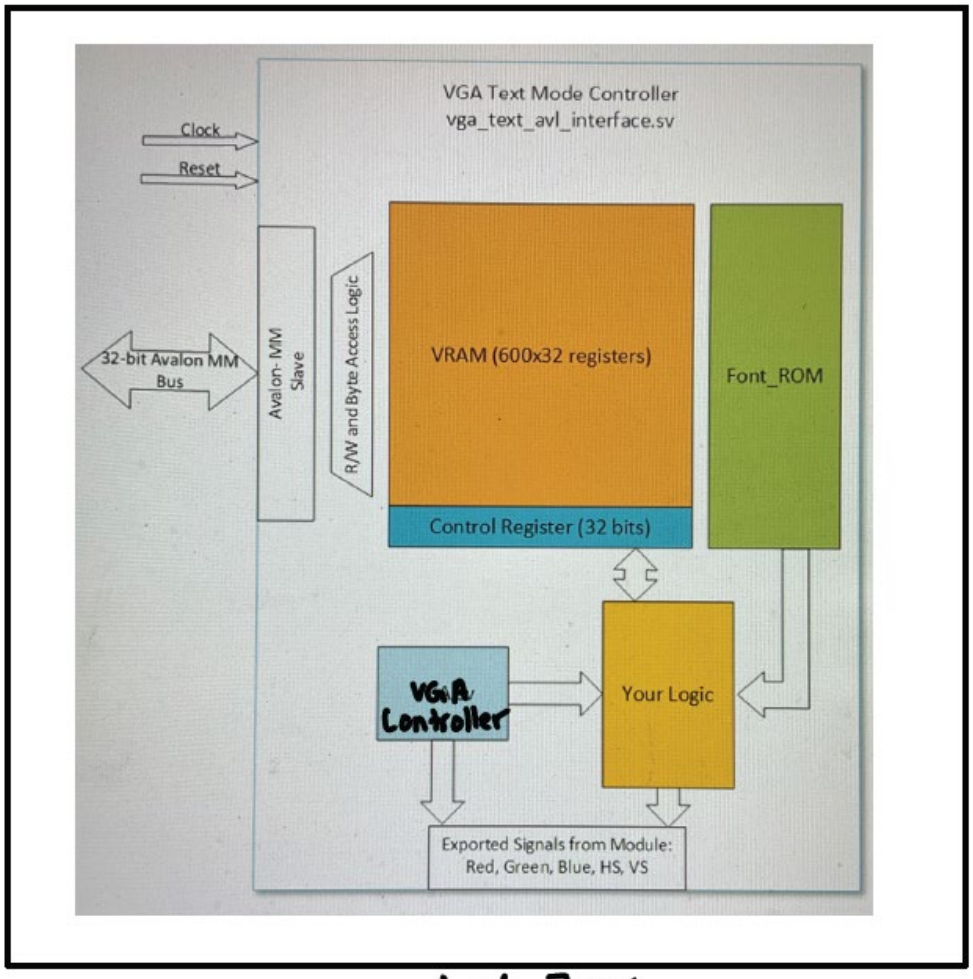
```
void setColorPalette (alt_u8 color, alt_u8 red, alt_u8 green, alt_u8 blue)
{
    //fill in this function to set the color palette starting at offset 0x0000 2000 (from base)
    alt_u8 row = color / 2;
    alt_u8 col = color % 2;
    alt_u32 myRow = vga_ctrl->myColors[row];
    alt_u32 num = 0;
    alt_u32 mask = 0;
    alt_u32 relevant_color = 0;

    if(col == 0) {
        num |= (red << 9);
        num |= (green << 5);
        num |= (blue << 1);
        mask = 0xffff7000;
        relevant_color = myRow & mask;
    } else {
        num |= (red << 21);
        num |= (green << 17);
        num |= (blue << 13);
        mask = 0x00001fff;
        relevant_color = myRow & mask;
    }

    vga_ctrl->myColors[row] = relevant_color | num;
    printf("%x \n", vga_ctrl->myColors[3]);
}
```

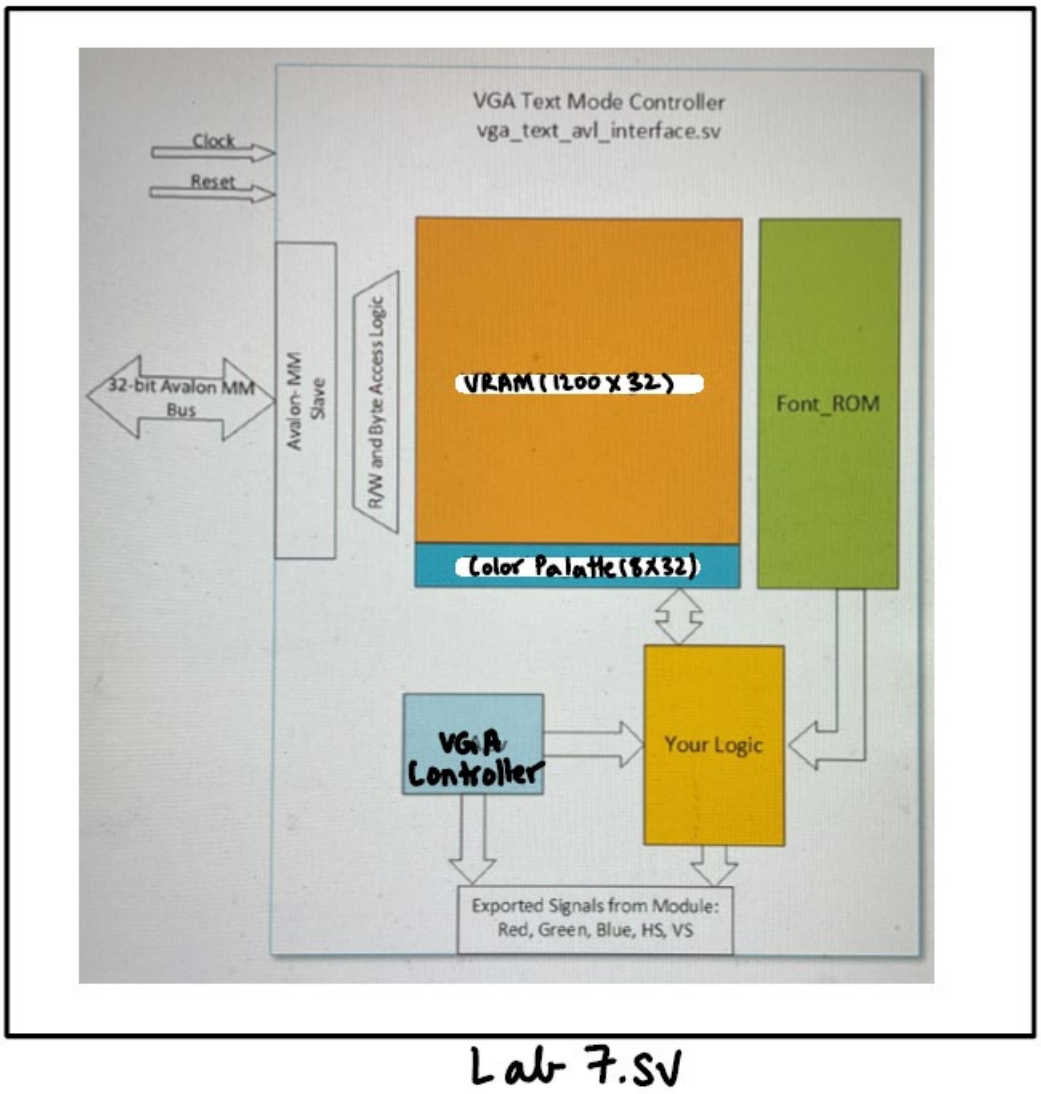
**Top Level Block Diagram:**

Week 1:




Lab 7.sv

Week 2:




**Module Descriptions:**

1. **Module:** vga_text_avl_interface.sv **(For Lab 7.1 only)**

   **Inputs:** CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [9:0] AVL_ADDR, [31:0] AVL_WRITEDATA

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

   **Description:** In this module we mainly initialize the VRAM registers, vga_controller, and font_rom. Based on DrawX and DrawY we access the VRAM registers and the font ROM to determine what RGB values should be outputted to the screen.

**Purpose:** This module basically decides what RGB values should be outputted based on DrawX and DrawY inputs from the VGA_Controller.

2. **Module:** vga_text_avl_interface.sv **(For Lab 7.2 only)**

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

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

   **Description:** In this module we mainly initialize the VRAM OCM, vga_controller, and font_rom. Based on DrawX and DrawY we access the VRAM module and the font ROM to determine what RGB values should be outputted to the screen. Here we additionally initialize the color palette, based on the color palette we decide the background and foreground values.

   **Purpose:** This module basically decides what RGB values should be outputted based on DrawX and DrawY inputs from the VGA_Controller and the color palette.

3. **Module:** VGA_controller.sv

   **Inputs:** Clk, Reset

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

   **Description:** This module handles the generation of the VGA signals. One of its keys features is to make a 25 Mhz clock from the 50 Mhz clock. This is done by using a always_ff block. Then this module keeps track of the horizontal and vertical pixel coordinates. It also resets them if they reach the end of horizontal and vertical pixel bounds. Another function is to generate the horizontal and vertical pixel sync pulse. Lastly, we use if else block to generate an active low blank signal.

   **Purpose:** It is the key component that generates the VGA signals. It provides us the DrawX and DrawY cordinates which are extremely important to draw anything on the screen. This module also generates a 25 MHz pixel clock. It provides info for the

blanking interval as well, so that we do not draw anything.

4. **Module:** font_rom.sv

   **Inputs:** [10:0] addr

   **Outputs:** [7:0] data

   **Description:** This is a Read Only Memory module. Here we store our 128 glyphs data for all the characters in form of 16 rows and 8 columns per glyph. We take an input address and return the respective 8-bit data of that row.

   **Purpose:** We use this module to draw the characters on the screen as it stores the pixel wise data for each glyph. We access a particular glyph structure and get its data as output.

5. **Module:** ram.v **(Lab 7.2 only)**

   **Inputs:** [11:0] address_a, [11:0] address_b, [3:0] byteena_a, clock, [31:0] data_a, [31:0] data_b, wren_a, wren_b

   **Outputs**: [31:0] q_a, [31:0] q_b

   **Description:** This module is a true dual port memory module that implements the On Chip Memory in our design. We generated this module via a IP wizard provided by Intel.

   **Purpose:** This module implements the On Chip Memory in our design. We use it to store our VRAM data. This is wizard generated IP by Intel.

6. **Module:** Lab7.sv

   **Inputs:** MAX10_CLK1_50, [1: 0] KEY, [9: 0] SW, [9: 0] LEDR, [7: 0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK, DRAM_CKE, DRAM_ADDR, [1: 0] DRAM_BA, [15: 0] DRAM_DQ

   **Outputs:** DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, VGA_R, VGA_G, VGA_B

   **Inout:** [15: 0] ARDUINO_IO, ARDUINO_RESET_N

**Description:** This is the top-level module of our design. This module just connects inputs and outputs to the instantiated modules in it. We instantiated Lab71soc and VGA controller here.

**Purpose:** This module just provides interface to connect the FPGA pins to the instantiated NIOS II and the VGA controller.

7. **Module:** Lab7.sv **(For Lab 7.1 & 7.2 only)**

   **Inputs:** spi0_MISO, clk_clk, [1:0] key_external_connection_export, reset_reset_n, usb_gpx_export, usb_irq_export.

   **Outputs:** sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export, [3:0] vga_port_blue, [3:0] vga_port_red, [3:0] vga_port_green, vga_port_hs, vga_port_vs.

   **Inout:** [15:0] sdram_wire_dq

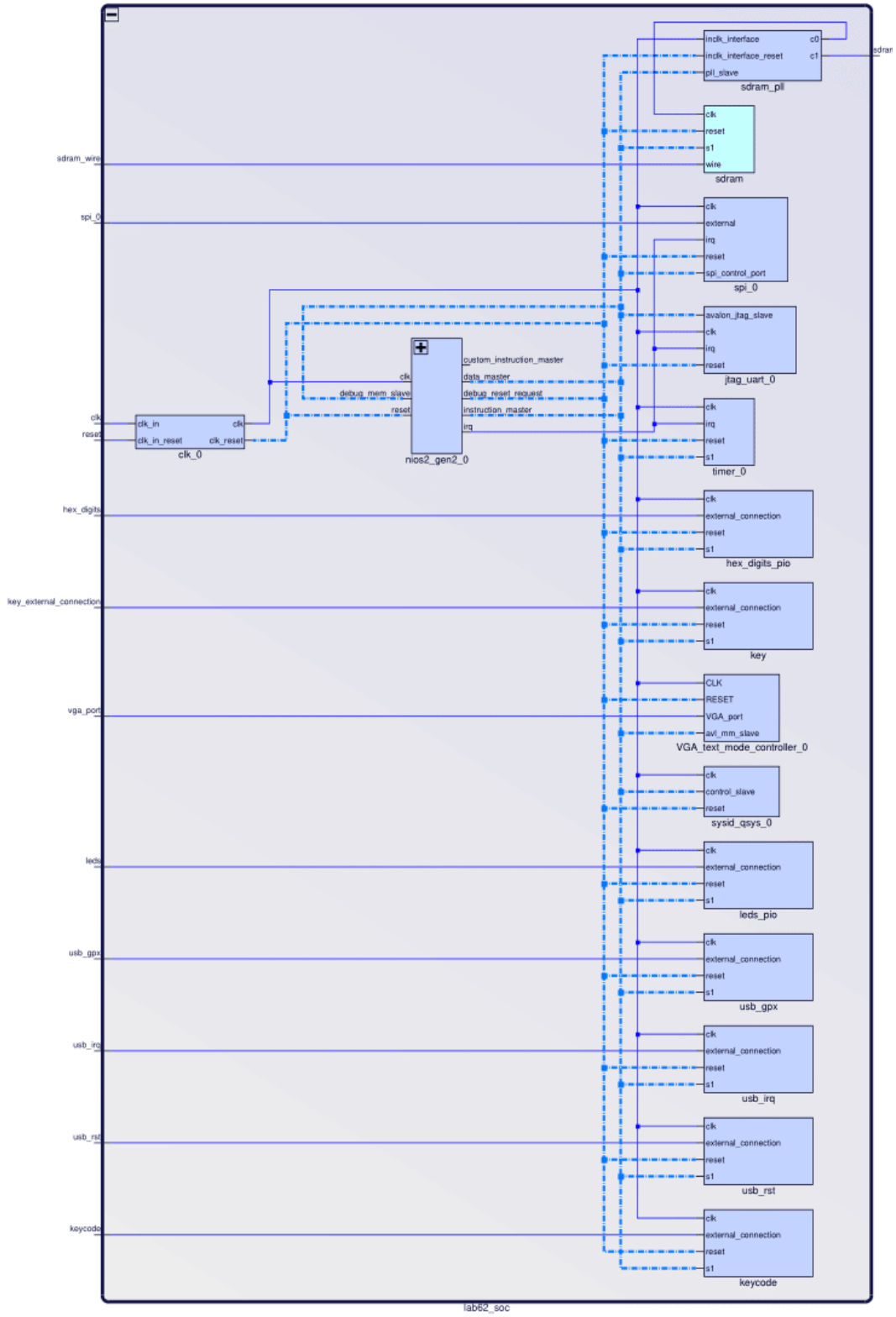
   **Description:** This a platform designer generated module. We have instantiated the following IPs in the platform designer which make up this module. This top-level module is so that we can connect inputs and outputs of these IPs. The IPs include: NIOS ll/e, OCM, PIOs, JTAG UART, SDRAM, PPL for SDRAM, SPI, TIMER. NIOS ll/e is our processor. JTAG UART is for debugging purposes that we use to debug the NIOS ll. SDRAM is the primary memory where our compiled code is stored as instructions. PLL is to accommodate the physical time offset due to the path difference between the SDRAM and the OCM. For Lab 7 we just additionally a custom IP for the VGA controller called the vga_text_alvon_interface. This module is memory mapped slave to the Avalon bus of the NIOS II which is responsible for generating the RGB signals to the VGA output. For lab 7.2, the only change that was made is to add the

RAM IP provided by intel to the platform designer so that the VRAM is no more implemented by registers.

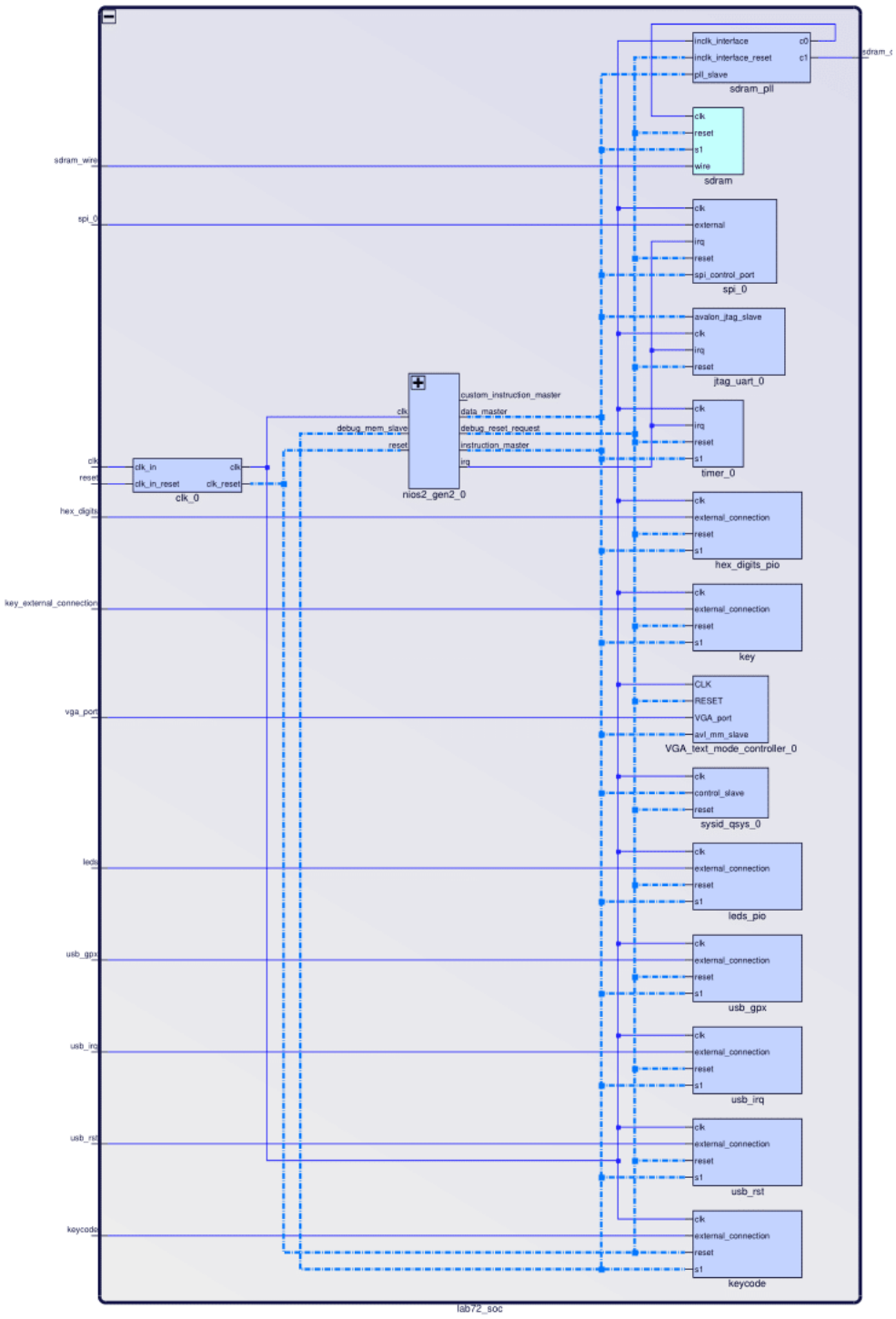
**Purpose:** This is the top-level module exported from the Platform designer for Lab 6.1 and implements all the IPs generated in the Platform designer to the FPGAs. These include the NIOS ll/e, PIOs, JTAG UART, SDRAM, PPL for SDRAM, and VGA_TEXT_AVALON_INTERFACE.

## System Level Block Diagram:

Week 1:



Week 2:


inclk_interface
inclk_interface_reset
pll_slave
c0
c1
sdram_pll
clk
reset
s1
wire
sdram
sdram_wire
clk
external
irq
reset
spi_control_port
spi_0
spi_0
avalon_jtag_slave
clk
irq
reset
jtag_uart_0
custom_instruction_master
clk
data_master
debug_mem_slave
debug_reset_request
reset
instruction_master
irq
nios2_gen2_0
clk
irq
reset
s1
timer_0
clk
reset
clk_in
clk_in_reset
clk
clk_reset
clk_0
hex_digits
clk
external_connection
reset
s1
hex_digits_pio
key_external_connection
clk
external_connection
reset
s1
key
vga_port
CLK
RESET
VGA_port
avl_mm_slave
VGA_text_mode_controller_0
clk
control_slave
reset
sysid_qsys_0
leds
clk
external_connection
reset
s1
leds_pio
usb_gpx
clk
external_connection
reset
s1
usb_gpx
usb_irq
clk
external_connection
reset
s1
usb_irq
usb_rst
clk
external_connection
reset
s1
usb_rst
keycode
clk
external_connection
reset
s1
keycode
lab72_soc

**Platform Designer:**

| Components | Description |
|---|---|
| Clk_0 | Used to generate clock for the FPGA, SDRM, as well as the MAX3421E. |
| Nios2_gen2_0 | The CPU layer that behaves like slc3 but is more robust. This CPU controls all functionality within the machine. It is responsible for parsing and acting on instructions that it fetches from memory. |
| Sdram | SDRAM is the external memory chip connected to the FPGA board. The NIOS II is stored here instead of the onchip memory. Instructions as well as data is fetched from here. |
| Sdram_pll | Since sdram is further away from the onchip memory and because it requires a clock signal, the PLL handles the timing offsets that are caused due to board layout. This component is very important as the SDRAM requires precise timings due to the fact that it is made up of capacitors. |
| Jtag_uart_0 | JTAG UART allows serial character communications streams between a host PC and the FPGA. In addition, the JTAG UART core is also connected to the Interrupt controller as transferring text over the console is slow and we don't want to slow down the CPU. |

| | |
|---|---|
| Timer_0 | This block handles the timer for the USB Port. The timer is essential to the USB driver in order to keep track any timeout that the USB needs. |
| Onchip_memory2_0 | Onchip memory is a small memory block that resides within the FPGA chip. Since onchip memory is very close to the actual FPGA, it is faster access times. |
| Sysid_qsys_0 | Used to check the system ID so that it can maintain the compatibility between software and hardware. It functions by giving a serial number which the software cross checks with the software. |
| VGA_text_mode_controller_0 | In this module we mainly initialize the VRAM registers, vga_controller, and font_rom. Based on DrawX and DrawY we access the VRAM registers and the font ROM to determine what RGB values should be outputted to the screen. It is used to implement the Avalon Memory Mapped bus. |

Note that not all blocks showed in the Quartus diagram are explained, because we don't use them in lab 7.

**Post Lab Questions Design & Resources and Statistics:**

Week 1:

| | |
|---|---|
| LUT | 32051 |
| DSP | 0 |
| Memory (BRAM) | 46079 bits |
| Flip-Flop | 21528 |
| Frequency | 67.76 MHz |
| Static Power | 98.46 mW |
| Dynamic Power | 232.45 mW |
| Total Power | 358.85 mW |

Week 2:

| | |
|---|---|
| LUT | 2565 |
| DSP | 0 |
| Memory (BRAM) | 192436 bits |
| Flip-Flop | 2528 |
| Frequency | 72.48 MHz |
| Static Power | 97.60 mW |
| Dynamic Power | 80.16 mW |
| Total Power | 198.83 mW |

The biggest difference between Lab 7.1 and 7.2 implementations is that Lab 7.1 uses 601 registers on the FPGA, while lab 7.2 uses OCM. This results in three main differences. Firstly Lab 7.1 uses a lot more LUTs as compared Lab 7.2, as each LUT comprises of a single register. While in contrast, Lab 7.2 has much higher BRAM memory usage as compared to Lab 7.1.

Secondly, since the registers are spaced out all across the FPGA it could take more time locating a specific register meaning that the overall design becomes slower as compared to Lab 7.2. Lastly, as expected, the power consumption of Lab 7.2 is also much smaller as compared to Lab 7.1 because we are using fewer number of LUTs.

**Conclusion:**

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

For this lab we successfully built a general purpose text-mode graphics controller. To do this we used glyphs stored in the font rom in conjunction with the Avalon bus to display certain characters on the VGA screen with background and foreground color. We were able to get both weeks working before our demos, so we faced no major issues.

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

This lab has been extremely beneficial in teaching us how to use display things on the VGA screen. For our final project we will use our knowledge about interfacing with VGA as well as NIOS II to draw things on the VGA screen and then use a neural network to recognize what digit is being written and then ultimately solve the equation.

*3) Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?*

The lab documents were pretty self-explanatory to help us do the initial set up, so we didn't have any problems there. However, the math required in the labs was not particularly trivial, so it would have helpful if the documents have us a more of an idea of how to go about doing the math that helps us index and access the glyphs from memory.