ECE 385

Fall 2022

Lab 7

VGA Text Mode Controller with Avalon-MM Interface

Zhuoyuan Li, Shihua Cheng

Tianhao Yu

Introduction

In Lab 7, we built a VGA text interface for the communication between the Nios II platform and the FPGA component that drives the VGA output. The VGA interface is a user-defined IP in the Platform Designer. It uses the Avalon Memory-Mapped Interface so that the user can access the memory inside the interface (registers for 7.1, on-chip memory for 7.2) through a struct in C code. The 640*480 screen is configured to contain 80*30 characters. The Nios II processor writes the color information of each character to the video memory in the VGA interface through the Avalon-MM Interface, and the VGA interface scans the video memory to map the color to each character on the screen, and outputs the VGA signal. The design for Lab 7.1 only supports per-screen color output (up to 2 simultaneous colors), while the design for Lab 7.2 supports per-character color output (up to 16 simultaneous colors).

The Platform Designer for Lab 7 is built from Lab 6. The following modules are preserved: *SDRAM, SDRAM_PLL, clk_0, sysid_qsys_0, nios2_gen2_0, jtag_uart_0*. The other modules involving USB were not used for this lab. A new module *VGA_text_mode_controller* is added. These modules will be introduced in the module description section.

Written Description of Lab 7 System

Week 1

Lab 7.1, we create a monochrome text mode graphics controller that connects the Avalon MM bus and output 80*30 characters through the VGA module. The design supports per screen color and inverted color, which means drawing the screen with inverted color. A with the vga_text_avl_interface is created. 600 registers, each with 1 word, are used in the VRAM and 1 register is used as the control register. A font-ROM is used to store the data of the character. In specific, if we have VRAM[0x15] = 0x0101038E, the font_ROM will output the 8 bit row of a character for 16 times with the input address provided by ”0x8E”, ”0x03”, “0x01”, and “0x01”, and the location that will be present on the VGA monitor is can be calculated by transferring the hex value to decimal: row = 21(0x15) * 4 / 80, and col = 21 * 4 % 80.

Embedded with 601 word registers VRAM, the VGA Text Mode controller reads from and writes to the VRAM as indicated by the AVL_READDATA, AVL_WRITE, AVL_ADDR, and AVL_WRITEDATA. By using the DrawX and DrawY signal from the VGA_controller, the corresponding address that should be read from the font_rom is calculated and inputted into the font_rom module, then the output’s next pixel that will be drawn will be xor with the iv to set its RGB, and finally outputted. LOCAL_REG enables us to get IVN and CODEN which will be necessary for calculating the address input of the font-rom.

The 601th register is the control register, whose [24:21] is foreground Red, [20:17] is foreground green, [16:13] is foreground blue, [12: 9] is background Red, [8:5] is background green, [4:1] is background blue.

The VRAM is implemented by a 600 word LOCAL_REG. When chip select is high and AVL_READ is high, which indicates a read operation, the AVL_READATA just gets the data from the local)_reg with the address provided by the AVL_ADDR. If the AVL_READ is low, the AVL_ADDR will be all 0. When chip select is high and AVL_WRITE is high, which indicates a write operation, the [3:0] byteenable chooses which bytes will be written, and then this byte of the Local register with address indicated by avl_addr's corresponding 8 bits will then be written by the [7:0] AVL_WRITEDATA, the data we want to write, and by these methods described above, the design enabled the read and write operation on the VRAM.

```
else begin
if (AVL_CS == 1'b1) begin
    if (AVL_READ == 1'b1) begin
        AVL_READDATA <= LOCAL_REG[AVL_ADDR];
    end
    else
        AVL_READDATA <= 32'h00000000;

    if (AVL_WRITE == 1'b1) begin
        if (AVL_BYTE_EN[0])
            LOCAL_REG[AVL_ADDR][7:0] = AVL_WRITEDATA[7:0];
        if (AVL_BYTE_EN[1])
            LOCAL_REG[AVL_ADDR][15:8] = AVL_WRITEDATA[15:8];
        if (AVL_BYTE_EN[2])
            LOCAL_REG[AVL_ADDR][23:16] = AVL_WRITEDATA[23:16];
        if (AVL_BYTE_EN[3])
            LOCAL_REG[AVL_ADDR][31:24] = AVL_WRITEDATA[31:24];
    end
end
end
```

```
always_comb begin
   posX = drawX[2:0];
   posY = drawY[3:0];
   col = (drawX >> 3);
   row = (drawY >> 4);
end

logic [1:0] byteSelect;
logic [9:0] regAddr;
logic [6:0] code;
logic iv;

always_comb begin
   byteSelect = (row * 80 + col);
   regAddr = ((row * 80 + col) >> 2);

   unique case (byteSelect)
      2'b11: begin
         code = LOCAL_REG[regAddr][30:24];
         iv = LOCAL_REG[regAddr][31];
      end
      2'b10: begin
         code = LOCAL_REG[regAddr][22:16];
         iv = LOCAL_REG[regAddr][23];
      end
      2'b01: begin
         code = LOCAL_REG[regAddr][14:8];
         iv = LOCAL_REG[regAddr][15];
      end
      2'b00: begin
         code = LOCAL_REG[regAddr][6:0];
         iv = LOCAL_REG[regAddr][7];
      end
   endcase
end
```

As for the font_rom, whose input is the [10:0] addr, the vga_controller will generate a DrawX and DrawY output, both of which have influence on the rom_addr we want to input into the vga_controller. The posY = drawY[3:0], and the posX = drawY[2:0], the pixel we got from the ROM is equal to rom_data[7-posX], where ROM data is the 7 bit output of the font_rom. The input of font_rom is rom_addr = {code, posY}, posY indicate specific row within the character, and code is the Glyph code from IBM Codepage 437 that refers to the 7-bit index to start address one of 128 characters. Code is obtained by the byte, chosen by byteSelect, of local register with the register address LOCAL_REG[regAddr], the byteSelect and regAddr are calculated by the col and row, which are obtained by the current drawing coordinate of the electron gun using the method below. This enables the design to get the pixel from the

font_ROM. col and row is the col and row of 80*30 VRAM. 640/8 = col, and 480/16 = 30. This transit drawX and drawY to the col and row of VRAM. The regAddr is just row*80 + col / 4 since there's 4 characters within each register, and byteselect is the reminder of such division. Byteselect indicates the position of the character within a register. Then CODEN is just the 7 bits indicated by the byteselecter and the iv is just the bit right in front of the CODEN.

The inverse color is managed by IVN for each CODEN within the VRAM, it could be got right in front of the code. We do a Xor between IV and the pixel that's about to be drawn, if the result is 1, the result RGB will be the foreground colors, while if the result is 0, which indicate either pixel is 0, but IV is 1, or the pixel is 1, but the IV is 0, the pixel will be drawn with RGB equals to the foreground colors.

Week 2

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

We deleted the 601 32-bit registers used in Week 1. For the eight color palettes, we used eight 32-bit registers. This is for parallel access, since the color mapping would access the pixel in VRAM and its corresponding color palette at the same time, and memory cannot be accessed in parallel. We used the megafunction to create an on-chip memory module. Following is the configuration of the on-chip memory.

Word length	32-bit
Number of words	4,096
Port type	Single
4-bit byte enable	Yes
Write enable	Yes
Read enable	No
Clocked output	No
Output when writing	Don't Care

The Avalon Memory-Mapped Interface is configured to wait for 2 cycles for both read and write so that the data can be processed correctly. Since the memory is both written and read by the Nios II processor, and is read by the VGA interface, the sharing of the on-chip memory must be configured carefully. We did not use a state machine to achieve memory sharing. Instead, we used a priority structure. Writing to the memory is of the highest priority. When the Nios II processor is writing to the memory, there should not be any output on the

screen, so ideally the “Output when writing” of the memory should be zero, but the writing time is very short so we used “Don’t care” for better efficiency. The “write enable” signal is set to “AVL_WRITE & AVL_CS”, so that write only happens when Nios II is writing to the VGA interface structure. For memory read, we did not use a read enable signal. Instead, we used a MUX for the address input for the memory:

```
206   always_comb begin
207       if (AVL_READ | wren)
208           addr = AVL_ADDR;
209       else
210           addr = regAddr;
211   end
```

AVL_ADDR is the input to the VGA interface provided by Nios II, and regAddr is the address of the register containing the character that is being written by the electron beam of the VGA controller, calculated from DrawX and DrawY. Since the if statement in SystemVerilog has priority, AVL_ADDR will be used as long as Nios II is accessing the memory. When Nios II finishes configuring the VRAM, the memory will always output the data of the current character being written for the color mapper to display the corresponding color. Using this design, for each VRAM update, the Nios II will first change the contents in the VRAM and the memory will be exclusive to Nios II (this will take a very short time), and then the memory will be exclusive to the VGA interface to provide color information.

Corresponding modifications to the Platform Designer IP

For the Platform Designer IP, the only modification needed was to change the address bits to 12, so that all the 12 bits in the on-chip memory are accessible.

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

For lab 7.1, four characters are stored in one memory location, while only two characters are stored in one memory location in lab 7.2. There are 80 characters per row, so the index of the character is calculated as (row * 80 + col). The variables row and col are calculated with “row = drawY >> 4” and “col = drawX >> 3”, since there are 16 pixels vertically and 8 pixels horizontally for each character. The address of the memory location is calculated as (row * 80 + col) >> 1 as opposed to (row * 80 + col) >> 2 in lab 7.1. The data is selected by the lowest bit of (row * 80 + col). If the lowest bit is 1, then the character is the upper 16 bits, otherwise it is the lower 16 bits.

Additional modifications necessary to support multicolored text

Once the 16-bit character information is obtained, it is decoded into the following data:

[15]	[14:8]	[7:4]	[3:0]

iv	code	fgd_idx	bgd_idx

“iv” is used to determine if the foreground and background colors are inverted, “code” is the code for the character being written, “fgd_idx” and “bgd_idx” are 4-bit indices that are used to represent the foreground color and the background color for the character. These indices are used to access the eight color palette registers. Since each register stores two colors, the address is obtained with (index >> 1). The lowest bit of the index is used to select the upper or lower color bits. If it is 1, the bits [24:13] are used to represent the RGB; otherwise, the bits [12:1] are used.

Additional hardware/code to draw palette colors

For color selection, some additional hardware logic is used:

```
if ((AVL_ADDR - 12'h800) >= 0 && AVL_WRITE == 1'b1) begin
    if (AVL_BYTE_EN[3])
        CTRL_REG[AVL_ADDR - 12'h800][31:24] <= AVL_WRITEDATA[31:24];
    if (AVL_BYTE_EN[2])
        CTRL_REG[AVL_ADDR - 12'h800][23:16] <= AVL_WRITEDATA[23:16];
    if (AVL_BYTE_EN[1])
        CTRL_REG[AVL_ADDR - 12'h800][15:8] <= AVL_WRITEDATA[15:8];
    if (AVL_BYTE_EN[0])
        CTRL_REG[AVL_ADDR - 12'h800][7:0] <= AVL_WRITEDATA[7:0];
end
```

The highest bit of AVL_ADDR is used to determine if Nios II is accessing the palette registers, and AVL_WRITE is used to make sure write does not happen on palette reads. Each bit of AVL_BYTE_EN is used to determine if the 8 bits in that location will be written to the palette registers.

On the Nios II platform, we completed the setColorPalette() function for the test program to run correctly. The main purpose of the function is to accept an 8-bit color input and three 8-bit values representing RGB. Although they are declared as 8-bit values, only the lowest 4 bits are used. The corresponding location in the palette registers should be set to the given RGB value. Following is the implementation of the function:

```
36 void setColorPalette (alt_u8 color, alt_u8 red, alt_u8 green, alt_u8 blue)
37 {
38     int n = color / 2;
39     int s = color % 2;
40
41     if (s == 1) {
42         // 0000000x 00000000 00000000 xxxxxxx0
43         vga_ctrl->VRAM[0x2000 + n*4 + 3] = (red >> 3);
44         vga_ctrl->VRAM[0x2000 + n*4 + 2] = ((red << 5) + (green << 1) + (blue >> 3)) % 0x0100;
45         vga_ctrl->VRAM[0x2000 + n*4 + 1] |= ((blue << 5) % 0x0100);
46         vga_ctrl->VRAM[0x2000 + n*4 + 1] &= (((blue << 5) + 0x1F) % 0x0100);
47     } else {
48         vga_ctrl->VRAM[0x2000 + n*4 + 1] &= !(0b00011111);
49         vga_ctrl->VRAM[0x2000 + n*4 + 1] += (red << 1);
50         vga_ctrl->VRAM[0x2000 + n*4 + 1] += (green >> 3);
51         vga_ctrl->VRAM[0x2000 + n*4 + 0] = ((green << 5) + (blue << 1)) % 0x0100;
52     }
53 }
```

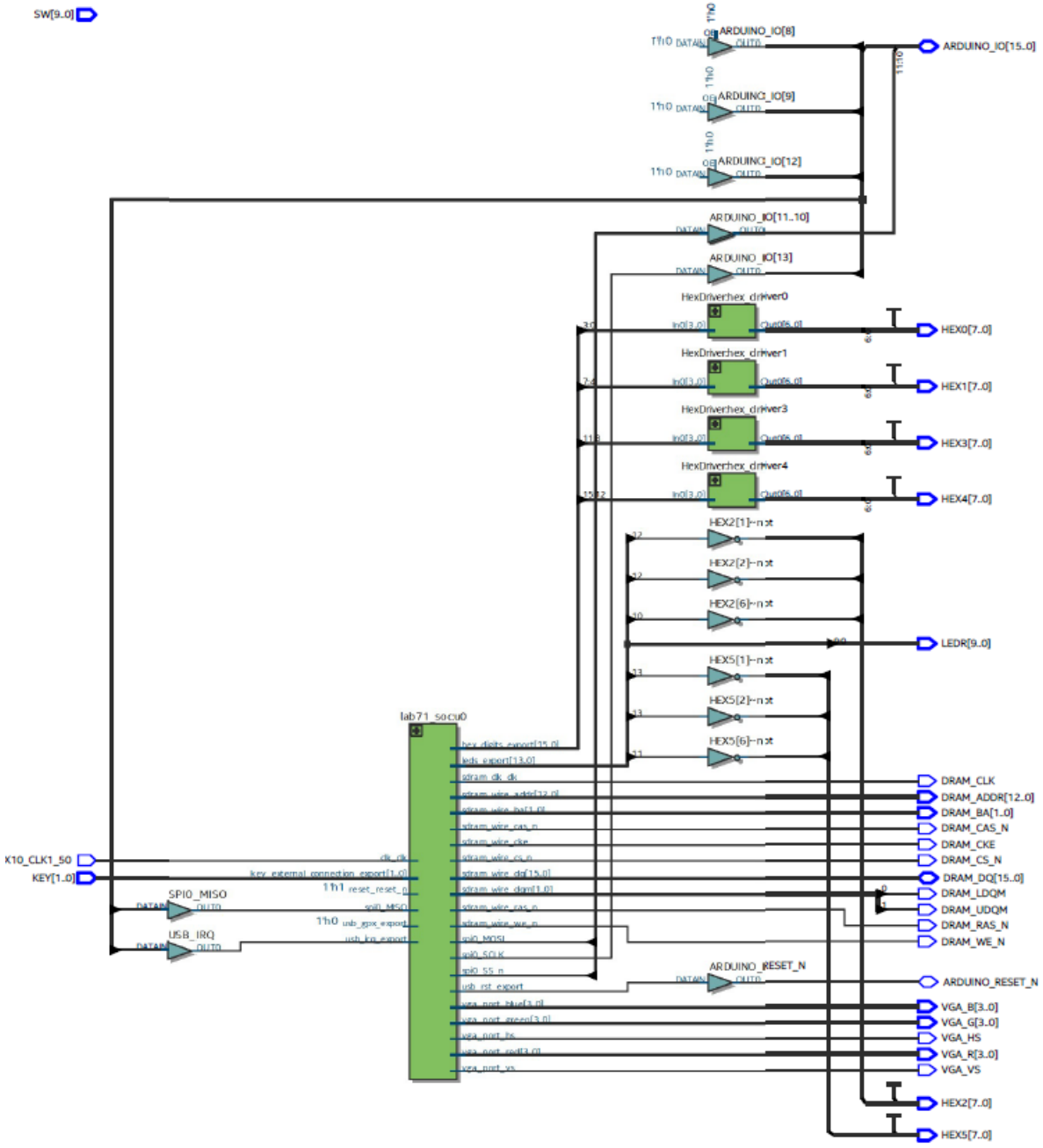
We hardcoded the function so that the RGB bits could fit in the correct location. The variable “n” is used to determine which register to write to, it is color/2 because each register contains two colors. The variable “s” is the lowest bit of “color” and is used to determine the location in the register to write to. The configuration of the color palette register is shown below:

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

We used bit operations AND and OR so that the RGB values can be set without changing the other bits in the register. Since Nios II is little-endian, the index is flipped when writing to the VRAM in Nios II. For example, to write to the upper 8 bits to register 0, the index will be VRAM[0x2000 + 3]. We aligned the bits as shown in the table above, and this function was able to write the correct RGB values to the correct location in the palette registers.

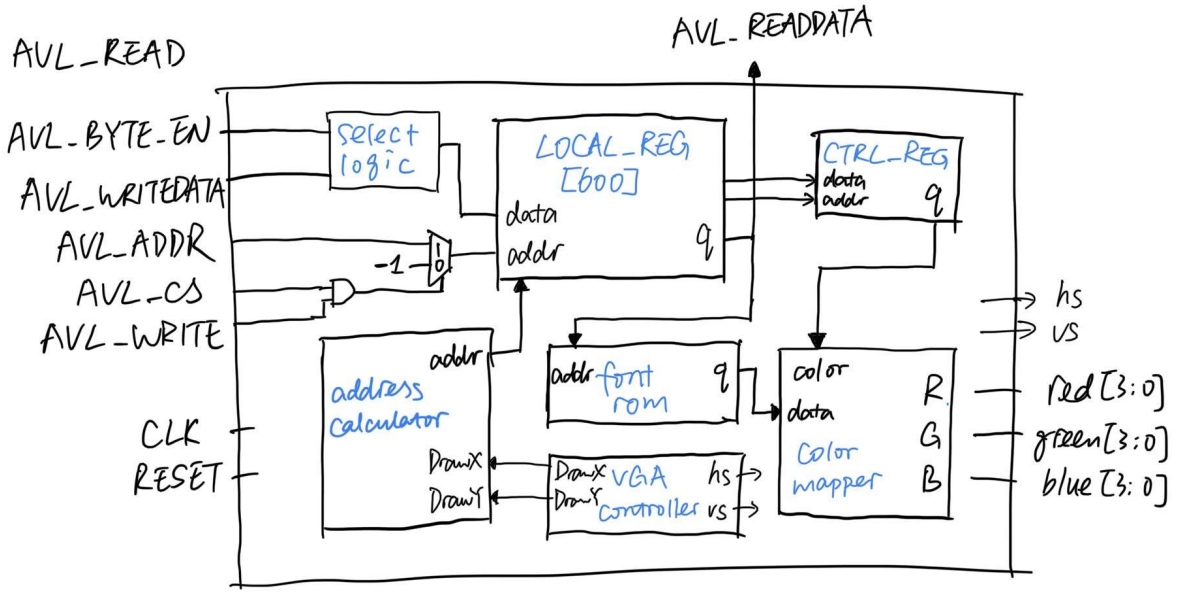
Block Diagram

The top-level block diagrams for both week 1 and week 2 are the same. The difference is inside the vga_text_avl_interface IP in the SoC. The top-level block diagram for both 7.1 and 7.2 is shown below. We did not use any state machine in Lab 7.2.

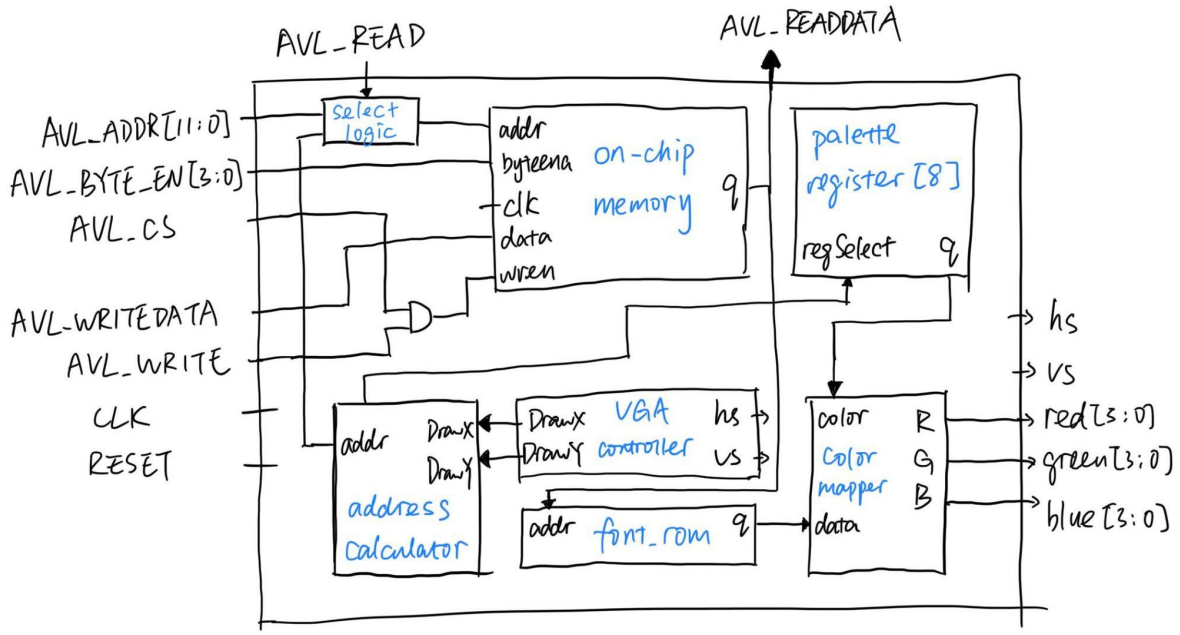


The difference between 7.1 and 7.2 is that in lab 7.1, we used 601 registers for VRAM, while in lab 7.2, we used the on-chip memory as VRAM to store 1,200 32-bit graphics information, and we used eight registers to store the color palette for parallel access. The diagrams of vga_text_mode_controller.sv for both week 1 and week 2 are shown below. The RESET logic for week 1 is to clear all 601 registers, and the RESET logic for week 2 is to clear the eight palette registers. The CLK input is connected to registers, memory, and the VGA controller.

Week 1



Week 2



Module Descriptions

.sv modules

font_rom.sv

input: [10:0] addr

output: [7:0] data

description: A ROM used to store the data of the characters that shall be drawn. There are 127 characters in total. All characters inside are stored in 16 rows, each containing 8 bits. Every time the output is one row of the data stored.

purpose: Used to store the data of all characters.

HexDriver.sv

input: [3:0] In0

output: [6:0] Out0

description: The dexdriver is used to convert a 4-bit unsigned integer (0-F) to hexadecimal form.

purpose: Convert a 4-bit unsigned integer (0-F) to hexadecimal form.

lab7.sv

input: MAX10_CLK1_50, [1: 0] KEY, [9: 0] SW, [15: 0] ARDUINO_IO, ARDUINO_RESET_N

output: [9: 0] LEDR, [7: 0] HEX0, [7: 0] HEX1, [7: 0] HEX2, [7: 0] HEX3, [7: 0] HEX4, [7: 0] HEX5, DRAM_CLK, DRAM_CKE, [12: 0] DRAM_ADDR, [1: 0] DRAM_BA, [15: 0] DRAM_DQ, 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, [15: 0] ARDUINO_IO, ARDUINO_RESET_N.

description: Used with lab7_soc, this serves as the top level. It is generally used to connect the components including the SDRAM, the VGA, the Adruino, the SPI, GPIO, the Led and the Hex and external components such as key.

purpose: Use to connect all the hardware components.

vga_controller.sv

input: Clk, Reset,

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

description: Literally the same VGA_controller as the one in lab 6.1. This is the module that controls the VGA signal output on the FPGA board, so that contents can be displayed on an external monitor. It has asynchronous reset, and it uses the 50MHz Clk input to generate the 25MHz pixel_clk signal. This 25MHz pixel clock provides 25MHz / 800 (horizontal) / 524 (vertical) = 59.6Hz refresh rate on the external monitor, which is the output vs. HS is the horizontal sync signal, it is equal to 25MHz/800 = 31.25KHz. Blank = 0 when the electron beam is in the blanking interval and the output color should be RGB=000. Sync is not used in this lab. DrawX is a number from 0-639, DrawY is a number from 0-479, they represent the coordinate of the current pixel being drawn.

purpose: This module is used to control the output VGA signal on the FPGA board so that contents can be displayed on an external monitor.

vga_text_avl_interface.sv

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

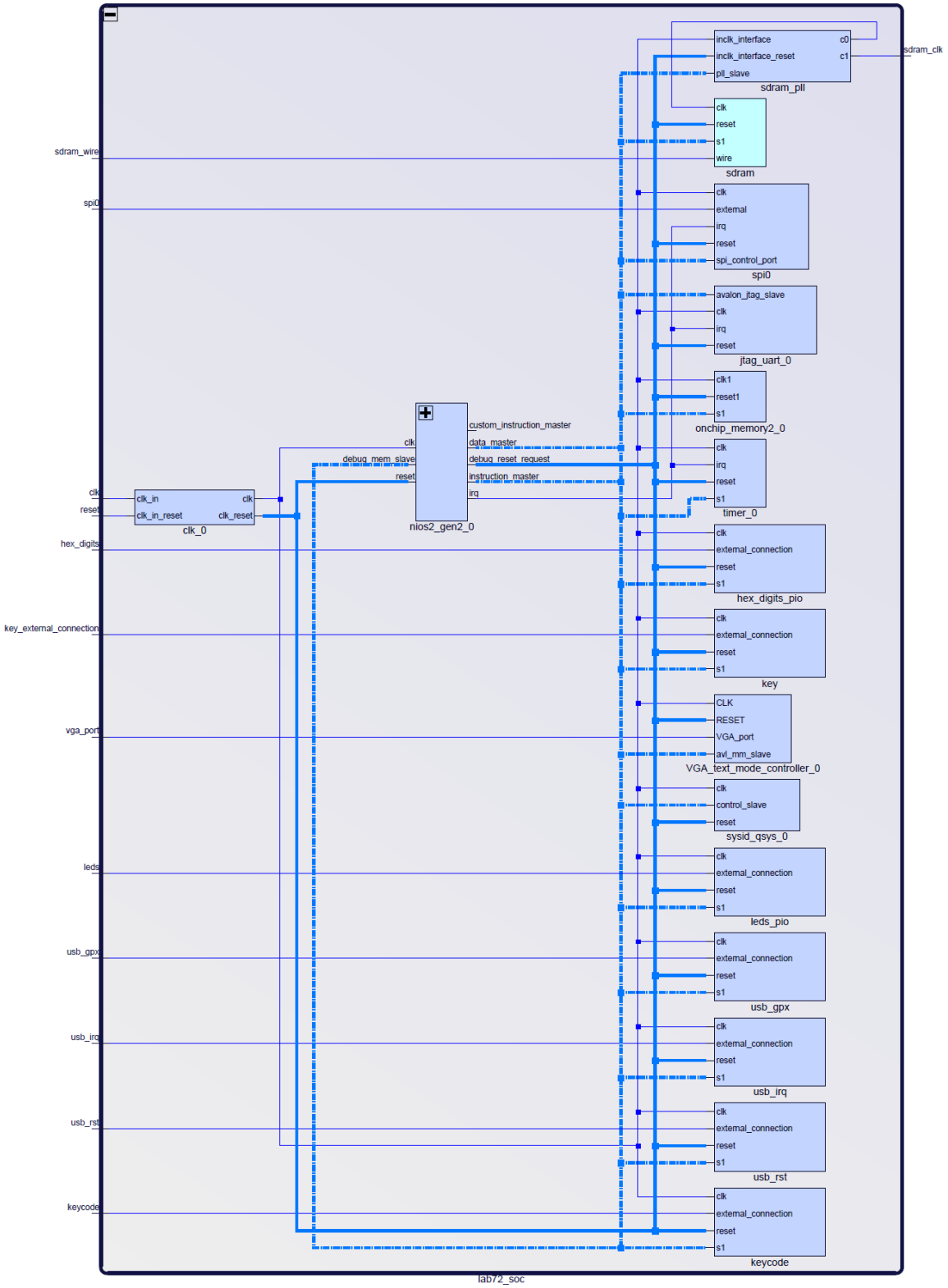
output: [31:0] AVL_READDATA, [3:0] red, green, blue, hs, vs

description: This is the interface module that enables the communication between the Nios II processor and the FPGA that drives the video output. It will only be active when AVL_CS is high. CLK is just clk_0. AVL_WRITE, AVL_READ are used to indicate whether a read or a write should be implemented. The AVL_BYTE_EN is used only in AVL_READ to indicate which type should be written. AVL_BYTE_EN[i] controls AVL_WRITEDATA[8*i + 7 : 8*i] to be written. A font_Rom is declared within to store the character data, and a VGA controller is declared to provide drawX and drawY. The red, green, blue, hs and vs outputs are provided to the VGA port. AVL_ADDR is the address which will be written to by the AVL_WRITEDATA. AVL_READDATA is the data that will be read from the AVL_ADDR during a read operation.

purpose: This is the interface module that enables the communication between the Nios II processor and the FPGA that drives the video output.

Platform Designer modules

The platform designer modules are adapted from Lab 6. Although modules related to USB such as SPI, timer and usb_* are present, they were not used for Lab 7. Following is the schematic view of the platform designer for both Lab 7.1 and Lab 7.2.



clk_0: This is the 50MHz clock generated on the DE-10 board. The clock is used by other modules in the Platform Designer. The reset signal can be assigned to a key on the board.

nios2_gen2_0: This is the Nios II/e processor used to execute C programs. The inputs Clk and reset are from the FPGA. The data_master is connected to the SDRAM to write and read

data needed for the program. The instruction_master is connected to the SDRAM to retrieve instructions.

sdram_pll: This is the sdram controller. It adds a 1ns phase to the input clock from FPGA and feeds the output clock that lags the FPGA clock to the SDRAM. This is needed to compensate for clock skew in board layout. The 1ns delay guarantees that all synchronous signals are stabilized at the rising clock of the SDRAM clock.

sdram: This is the Synchronous Dynamic RAM used by the Nios II processor to store data and instructions to execute the program. It has 512 Mbits of size.

sysid_qsys_0: This module produces a serial number which will be checked by the software loader before software execution. It prevents loading software onto an FPGA with incompatible Nios II configuration.

jtag_uart_0: This module is used for debugging. When the Nios II processor executes "printf" functions, this module allows it to print to the console on the computer.

VGA_text_mode_controller_0:

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

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

Description: This is the interface module between the Nios II processor and the FPGA that drives the video output. It will only be active when AVL_CS is high. CLK is the 50MHz clock from *clk_0*. [11:0]AVL_ADDR is the 12-bit address for read/write. Each word has 32 bits. When AVL_WRITE is high, AVL_WRITEDATA will be written to the location indicated by AVL_ADDR. When AVL_READ is high, AVL_READDATA will be the data stored in the location AVL_ADDR. [3:0] AVL_BYTE_EN is the byte enable signal. AVL_BYTE_EN[i] controls AVL_WRITEDATA[8*i + 7 : 8*i] to be written. If it is high, the byte will be written; otherwise it will not be written. The VGA controller inside this module will display the data stored in the memory on the screen. The red, green, blue, hs and vs outputs are provided to the VGA port.

Purpose: This is the interface between the Nios II processor and the VGA driver. Its memory contents can be accessed and changed by programs running on Nios II, and it will drive VGA signals to display these contents.

The other modules were not used in Lab 7.

Design Resources and Statistics

Week 1:

LUT	32,528
DSP	0
BRAM	11,392 bits
Flip-Flop	21,789
Max Frequency	66.51 MHz
Static Power	97.58 mW
Dynamic Power	283.58 mW
Total Power	403.04 mW

Week 2:

LUT	4,548
DSP	0
BRAM	142,464 bits
Flip-Flop	2,729
Max Frequency	73.63 MHz
Static Power	96.57 mW
Dynamic Power	72.51 mW
Total Power	191.17 mW

Conclusion

Our design for both Lab 7.1 and Lab 7.2 worked as expected. There was a small bug when we were doing Lab 7.2. The monitor would display the correct text, but there were some glitch pixels that weren't displayed correctly. It turned out that we had used an extra flip-flop between the on-chip memory and the VGA text interface, and the issue was caused by the delay of the flip-flop. We then removed the flip-flop and the bug was fixed.

One potential extension of this design is to incorporate the color control in this lab with the USB interface in Lab 6, so that the on-screen colors can be changed according to user input. Another potential extension is to modify the font_ram module to display more complicated patterns instead of text. These extensions would be useful for the final project since we will need to render each frame according to user input, and the patterns will be more complicated.

There was nothing ambiguous or incorrect in the lab manual.