ECE 385

Fall 2021

Experiment #6

VGA Text Mode Controller with Avalon-MM Interface

Michael Grawe, Matthew Fang
ABE
Abigail Wezelis

Index

Introduction

The Purpose of Lab 7 was to learn how to interface in different ways with a VGA monitor and write text onto a VGA monitor. Through both labs 7.1 and 7.2, this was mainly accomplished with the use of an Avalon Bus, which was used to write color text onto the VGA display: monochrome for week one, and full color for week two. The data to be written to the screen is stored in VRAM, which is implemented in registers for 7.1. This implementation had to be changed to implement the palette and write in full color for 7.2, since the number of logical elements is not sufficient to support the VRAM doubling in size. Because of this, the VRAM is implemented in On-Chip-Memory to make room on the FPGA.

Written Description of Lab 7 System

Week1

The week one system for our Lab 7 design implemented a monochrome text display. The screen was divided up into 80 columns with each column having 30 rows of sprite cells. Each sprite cell was able to be filled in with a sprite. This was done by using 600 VRAM locations, implemented as registers, to hold data containing the code for four sprites to be drawn into 4 sequential locations on the screen. A general layout of the contents of a VRAM address is shown below.

Bit	31	30-24	23	22-16	15	14-8	7	3-0
Function	IV3	CODE3	IV2	CODE2	IV1	CODE1	IV0	CODE0

To get data to and from this VRAM location, memory must be written and read from using an AVALON bus. The implementation of the bus was done both in Platform Designer and in a System Verilog file. A byte enable signal was included with the AVALON bus so that one byte at a time could be written to. This is quite important with the layout of the VRAM addresses, since sprites to be drawn at a certain location are stored in 1 byte of data. By overwriting one byte at a time, each character can be modified individually, rather than rewriting a whole memory location. In the Platform designer, the AVALON bus is contained within the VGA text mode controller IP. This IP implemented not only the bus, but the also the processing of the RGB signals sent from our System Verilog logic using VGA_port. The logic to write into the VGA registers was handled using the AVL signals from/to our AVALON bus. If the write signal was high, then the byte enable was checked to see which bytes of write data to write into the AVL_ADDR (which pointed to a location in VRAM registers). If read was high, all of the data from the VRAM registers was read, since byte enable is not necessary for a read operation.

One of the more logic intensive portions of Lab 7 week 1 was determining whether a specific pixel on screen should be a 1 or a 0. The only input that our logic had to determine this was the (DrawX, DrawY) coordinates from the VGA controller. From these

two coordinates, the row of the sprite being drawn was determined by taking the Draw Y signal divided by 16, and the column was given by Draw X divided by 8. These calculations are correct because the sprites are all of size 16x8. The byte address of the sprite to be drawn was then determined by multiplying the row by 80 and adding it to the column value. This is necessary since there are 80 columns per row. From the byte address, the actual VRAM address is obtained by dividing the byte address by 4, since there are four sprite bytes per VRAM address. It should be noted here that all division were implemented by 'bit-shifting' since all the divisors are powers of two. This bit shifting just cuts off the number of shifted bits. After obtaining the VRAM address, we index into the correct sprite (byte) in VRAM by using the lower two bits of or byte address, which were previously unused when choosing VRAM address. Now that the sprite code is obtained, it is multiplied by 16 and added with the lower 4 bits of Draw Y to index into to the correct row of our font_rom file that contains the bitwise data of how to draw each sprite. Finally, the lower 3 bits of Draw X index to the correct bit of the font_rom row and determine whether the pixel should be a 1 or a 0.

The final step to draw monochrome on screen is to use this 1 or 0 from font_rom to determine the color to be drawn on the VGA monitor. This is done using the control register and the inversion bit seen in the VRAM address. If the Inversion bit is 1, then the 1 or 0 from font rom is inverted to become a final pixel value. This pixel value is then used to determine whether the background (0) or foreground (1) color is to be drawn. The red, green and blue signals are connected to the corresponding background and foreground signals in the control register and chosen based on the final pixel value to draw the monochrome characters on screen.

Week 2

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

During the second week of this lab, utilization of on-chip memory based VRAM replaced register-based VRAM. While registers have as many input and output ports as there are storage bits, memory has a fixed amount of on-chip memory ports. M9K blocks were utilized in this lab, which only has two ports, which may read or write. To compensate for the limited ports, the color palettes stored in memory were also stored as registers, meaning one port could be used to read the words which contained the character code, foreground index, and background index. The character code would then be routed to font_rom to decide the shape of the character while the foreground and background indices could be read from registers, all of which do not need to access memory, solving the port limitations.

Corresponding modifications to the platform designer IP

The only modification to the IP was the increase in size of the address port from 10 to 12. In week 2, the graphics controller still supported 80 columns by 30 rows, for a total of 2400 characters. While this only needed 600 words for week 1, because each word could only support 2 characters due to the additional foreground and background color information, 1200 words are needed for week 2. While an address width of 11 would allow access to 1200 memory addresses, 12 was used instead for color palette implementation. Set color palette was implemented in a way to write to the base memory address + 2048. This allows for the hardware to check for a logical 1 in the 12th

address bit, which signifies writing or reading from the color palette and the value can be saved into a register for easier access. The extra space would be unused but still be reserved for Platform Designer.

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

In addition, the write wait was also increased to 1 to account for writing to memory instead of registers. In week 1, no delay was needed because reading from registered requires is much faster than reading from memory.

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

New indexing is needed to account for the fact that only two characters instead of four characters are stored in a word. The same calculator to calculate which column, row and address, which is defined as column + (row * 80). But instead of dividing this value by four to calculate the VRAM address, this week's address has to be divided by two to find the VRAM address. Then, instead of using the least two significant bits of address to determine which character from the word is needed, only the least significant bit of address is needed this week, since there are only two characters in each word.

Additional modifications necessary to support multi colored text

Additional logic was needed to index the palettes to control the multicolored text, since two colors were stored in each register. By looking at the least significant bit of the foreground and background indexes, the position of each color in the register could be determined and the corresponding red, green, and blue values could be passed to the VGA.

Additional hardware/code to draw palette colors

```
void textVGAColorScreenSaver()
{
    //This is the function you call for your week 2 demo
    char color_string[80];
    int fg, bg, x, y;
    textVGAColorClr();
    //initialize palette
    for (int i = 0; i < 16; i++)
    {
        setColorPalette (i, colors[i].red, colors[i].green, colors[i].blue);
    }
    while (1)
    {
        fg = rand() % 16;
        bg = rand() % 16;
        while (fg == bg)
        {
            fg = rand() % 16;
            bg = rand() % 16;
        }
        sprintf(color_string, "Drawing %s text with %s background", colors[fg].name, colors[bg].name);
        x = rand() % (80-strlen(color_string));
        y = rand() % 30;
        textVGADrawColorText (color_string, x, y, bg, fg);
        usleep (100000);
    }
}
```

The textVGAColorScreenSaver function is the backbone to drawing text on the VGA. The function first clears the memory, which creates a blank screen. Then, setColorPalette initializes a list of predetermined colors into VRAM. Finally, in an infinite loop, the function randomly chooses a location to write a line of text with two randomly selected foreground and background 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)
    //volatile unsigned int palette = (unsigned int)0x8002000;

    alt_u8 idx = color / 2;//automatically rounds down
    alt_u32* coloridx = (alt_u32*)(vga_ctrl) + 2048 + idx;
    alt_u32 temp = 0x0;

    if(!(color % 2)){ //resets
        printf("reset\n");
        *coloridx = temp;
    }
    printf("color: %x\n", color);

    alt_32 redtemp = red << 9;
    alt_32 greentemp = green << 5;
    alt_32 bluetemp = blue << 1;
    temp = redtemp + greentemp + bluetemp;
    if(color % 2){
        temp = temp << 12; //if second value then left shift 12 times if this is the second
    }
    printf("temp: %x\n", temp);
    *coloridx += temp;
}
```

SetColorPalette initializes the colors into the correct memory locations. Since two colors fit into every address, extra logic is needed. The first if statement checks if this is the first color to be inputted, if so, it clears the memory in preparation. Then, the red, green, and blue values predetermined in the color struct are shifted accordingly to occupy the 13 lower bits. The color is then checked once more for whether it is the first or second color to be imputed. If the color index is odd, that means it will occupy bits 24-13 and the bits are shifted an additional 12 bits before being inserted into memory.

```
static struct COLOR colors[]={
    {"black",          0x0, 0x0, 0x0},
    {"blue",           0x0, 0x0, 0xa},
    {"green",          0x0, 0xa, 0x0},
    {"cyan",           0x0, 0xa, 0xa},
    {"red",            0xa, 0x0, 0x0},
    {"magenta",        0xa, 0x0, 0xa},
    {"brown",          0xa, 0x5, 0x0},
    {"light gray",     0xa, 0xa, 0xa},
    {"dark gray",      0x5, 0x5, 0x5},
    {"light blue",     0x5, 0x5, 0xf},
    {"light green",    0x5, 0xf, 0x5},
    {"light cyan",     0x5, 0xf, 0xf},
    {"light red",      0xf, 0x5, 0x5},
    {"light magenta",  0xf, 0x5, 0xf},
    {"yellow",         0xf, 0xf, 0x5},
    {"white",          0xf, 0xf, 0xf}
};
```

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

When values are written and read from memory, the hardware has an additional check for whether the address corresponds to a color palette. If so, the data would instead be written or read to or from a register.

```
always_ff @ (posedge CLK)
begin //write into palette registers
   if(AVL_CS && AVL_WRITE && AVL_ADDR[11])
   begin
      palette[AVL_ADDR[10:0]] <= AVL_WRITEDATA;
   end
   else if(AVL_CS && AVL_READ && AVL_ADDR[11]) //need a valid rea/write for nios to access memory
   begin
      AVL_READDATA <= palette[AVL_ADDR[10:0]];
   end
```

As mentioned above, because of the limited on-chip memory ports, the color was then read from registers instead of memory. These values are then stored in foreground and background logic until the inverted bit is checked, as in week 1, before the actual red, green, and blue values are determined from the foreground and background logic and sent to the VGA.

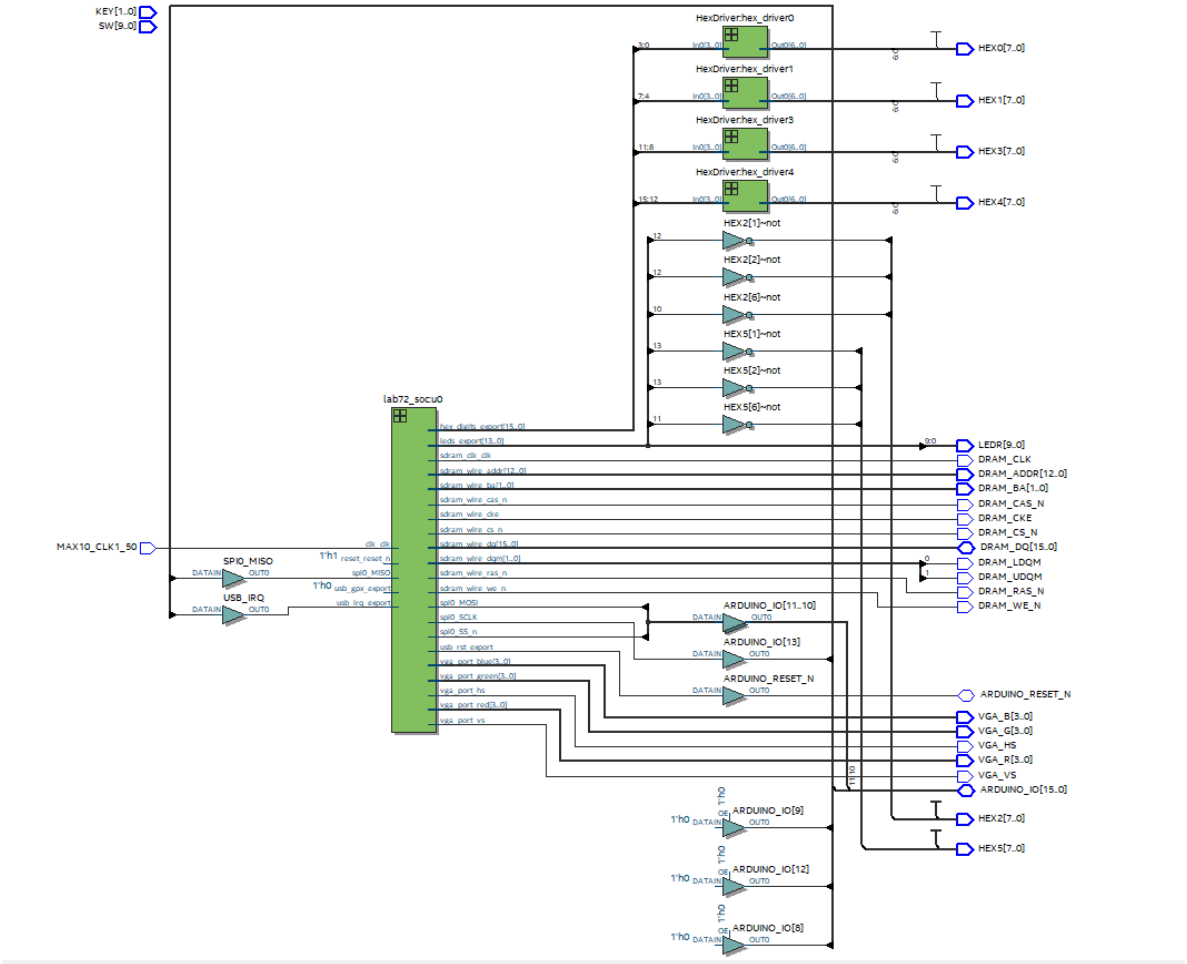
```
always_comb
begin //fgd, bgd from pallete initialization
   fgdidx = char[7:4];//char[7:4] get fgdidx and char[3:0] gets bgdidx (one number out of 16)
   bgdidx = char[3:0];
   if(fgdidx[0] == 1) //if one it means it was odd and we need [24:21][20:17][16:13]
   begin
      fgd_red = palette[fgdidx[3:1]][24:21];
      fgd_green = palette[fgdidx[3:1]][20:17];
      fgd_blue = palette[fgdidx[3:1]][16:13];
   end
   else
   begin
      fgd_red = palette[fgdidx[3:1]][12:9];
      fgd_green = palette[fgdidx[3:1]][8:5];
      fgd_blue = palette[fgdidx[3:1]][4:1];
   end

   if(bgdidx[0] == 1) //if one it means it was odd and we need [24:21][20:17][16:13]
   begin
      bgd_red = palette[bgdidx[3:1]][24:21];
      bgd_green = palette[bgdidx[3:1]][20:17];
      bgd_blue = palette[bgdidx[3:1]][16:13];
   end
   else
   begin
      bgd_red = palette[bgdidx[3:1]][12:9];
      bgd_green = palette[bgdidx[3:1]][8:5];
      bgd_blue = palette[bgdidx[3:1]][4:1];
   end
end
```

```
void textVGADrawColorText(char* str, int x, int y, alt_u8 background, alt_u8 foreground)
{
    int i = 0;
    while (str[i]!=0)
    {
        vga_ctrl->VRAM[(y*COLUMNS + x + i) * 2] = foreground << 4 | background;
        vga_ctrl->VRAM[(y*COLUMNS + x + i) * 2 + 1] = str[i];
        i++;
    }
}
```

The textVGADrawColorText function is in charge of actually writing characters into memory based on a given x and y. Once written in memory, the next time the VGA is drawing the pixels corresponding to those x and y values, the hardware reads and draws the updated character.

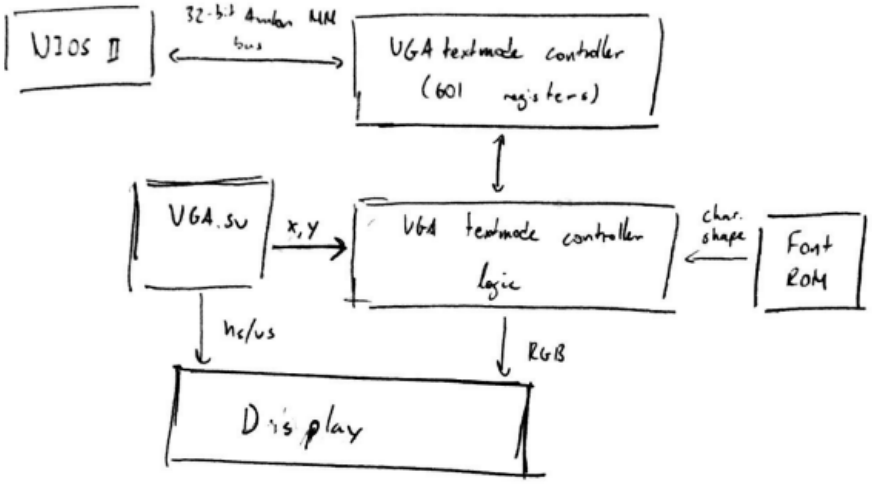
Block Diagram

The following is the top level for both weeks of lab 7

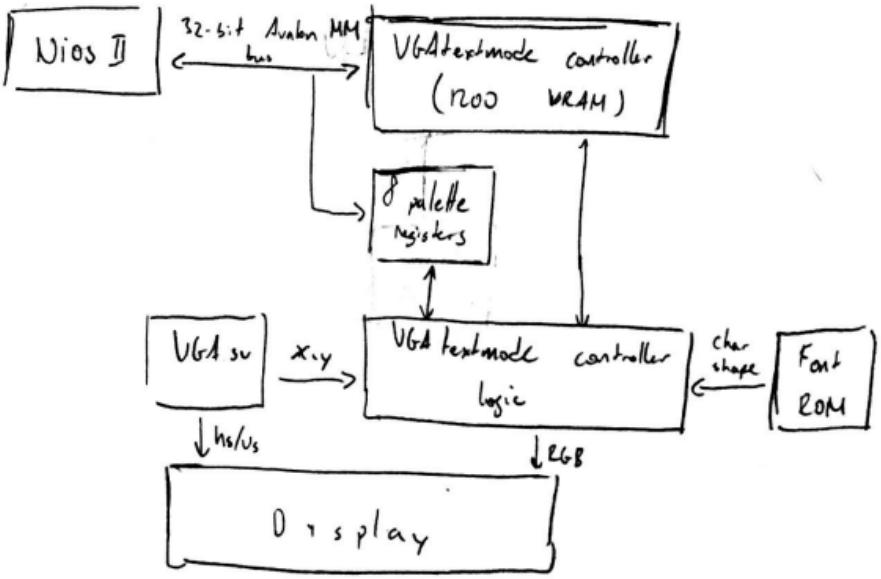


Both have the same top level diagram and very similar block diagrams. The only difference between the two is that instead of only interacting with 601 registers in week 1, week 2 writes to VRAM and 8 palette registers instead. The changes in the output of each week, the monochrome text display and color text display comes from the functions written in the NIOS II.

Week 1


Nios II
32-bit Avalon MM bus
VGA textmode controller (601 registers)
VGA_SU
x,y
VGA textmode controller logic
char. shape
Font ROM
hs/vs
RGB
Display

Week 2


Nios II
32-bit Avalon MM bus
VGA textmode controller (no VRAM)
8 palette registers
VGA_SU
x,y
VGA textmode controller logic
char. shape
Font ROM
hs/vs
RGB
Display

Written Description of all .sv Modules

Module: lab7

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

Outputs: [9:0] LEDR,
[7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
DRAM_CLK, DRAM_CKE, DRAM_LDQM, DRAM_UDQM,
DRAM_CS_N, DRAM_WE_N,
DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS
[3:0] VGA_R, VGA_G, VGA_B,
[12:0] DRAM_ADDR,
[1:0] DRAM_BA

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

Description: This is the top-level module used. Within this module, lab7_soc is instantiated, the Hex drivers are connected to the connect signals, and the Arduino signals are connected to the corresponding USB signal.

Purpose: The purpose of lab62 is to program the FPGA with the NIOS II processor and appropriate PIO's described in the Platform designer

Module: lab7_soc/synthesis/lab7_soc

Description: These are the hardware components created in the platform designer.

Purpose: These files allow the interaction with the keyboard and monitor peripherals. They also connect the NIOS II CPU for our design.

Module: vga_text_avl_interface

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

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

Description: vga_text_avl_interface implements the VRAM, the logic to write/read from the VRAM through the AVALON bus, as well as the algorithm to draw the characters from VRAM and font_rom.

Purpose: The purpose of color mapper is to handle read/write operation into VRAM, whether it be on-chip or registers, and to translate the contents of memory into colors on screen.

Module: VGA_controller
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync
[9:0] DrawX, DrawY

Description: VGA_controller handles vertical and horizontal sync while outputting DrawX and DrawY signals to represent the part of the screen to be drawn, which is analyzed by other modules to determine what pixel should be drawn.

Purpose: The purpose of VGA_controller is to move the virtual electron gun across the screen and handle the vertical and horizontal sync in order to draw pixels.

Module: Hex Driver
Inputs: [3:0] In0
Outputs: [6:0] Out0

Description: For the purposes of Lab 7, the Hex Drivers are connected but unused.

Purpose: The purpose of hexdriver is to display the keycode of the most recent key pressed.

Module: lab7.sdc

Description: This module sets output delays for precise time signals coming from dram and false paths for the less time sensitive signals from the switches, keys, and leds.

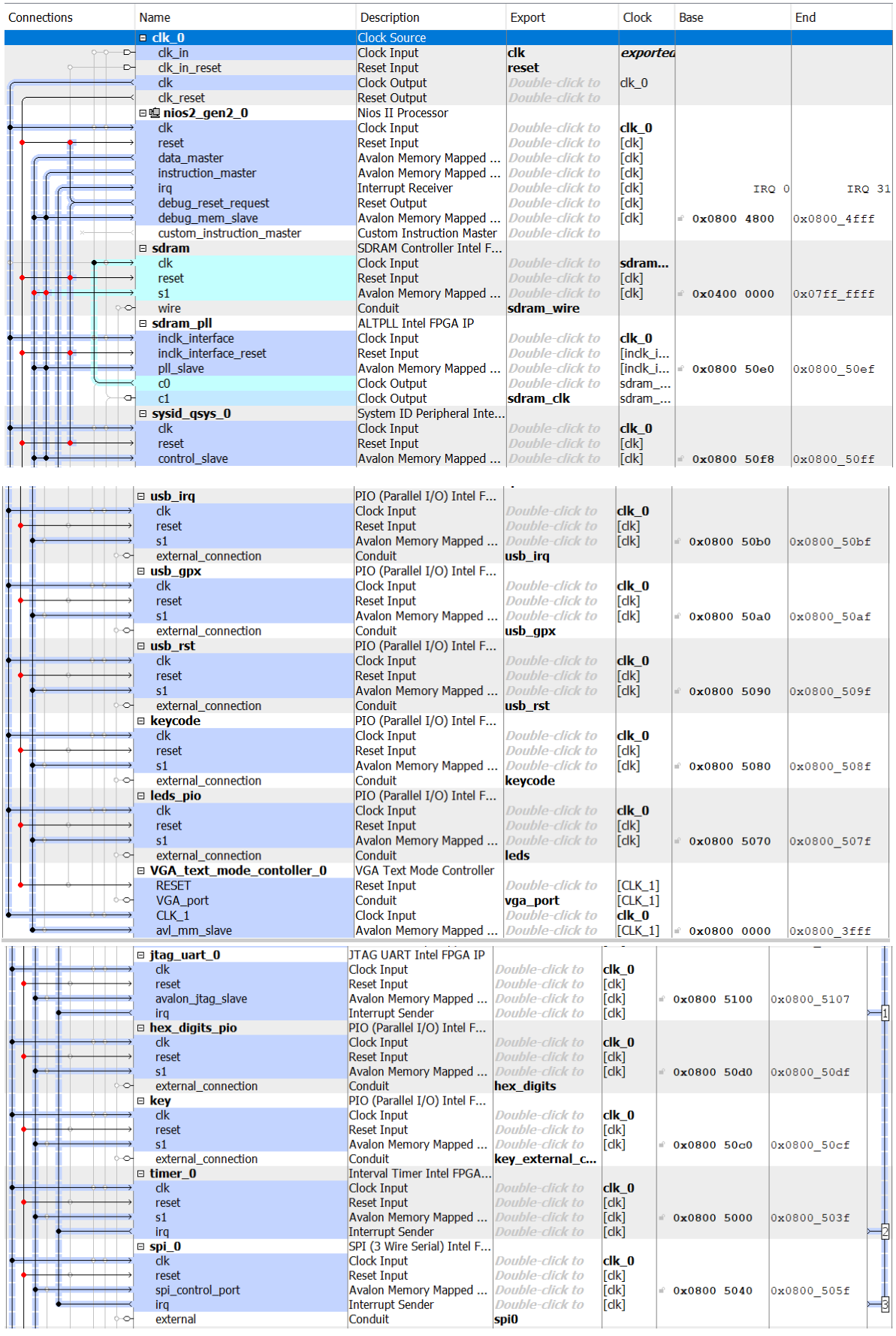
Purpose: The purpose of this timing file is to ensure external signals are stable before measurement.

Module: ram.v
Inputs: [3:0] byteena_a
Clock, wren
[31:0] data
[10:0] rdaddress, wraddress
Outputs: [31:0] q

Description: This module comes from a Quartus Mega-function to implement on-chip memory. It implements a two-port, one read one write memory.

Purpose: The purpose of this module is to act as VRAM in On-chip memory for Lab 7 week 2.

Platform Designer View of Modules



Connections	Name	Description	Export	Clock	Base	End
	clk_0	Clock Source				
	clk_in	Clock Input	**clk**	***exported***		
	clk_in_reset	Reset Input	**reset**			
	clk	Clock Output	*Double-click to*	clk_0		
	clk_reset	Reset Output	*Double-click to*			
	nios2_gen2_0	Nios II Processor				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	data_master	Avalon Memory Mapped ...	*Double-click to*	[clk]		
	instruction_master	Avalon Memory Mapped ...	*Double-click to*	[clk]		
	irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31
	debug_reset_request	Reset Output	*Double-click to*	[clk]		
	debug_mem_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 4800	0x0800_4fff
	custom_instruction_master	Custom Instruction Master	*Double-click to*			
	sdram	SDRAM Controller Intel F...				
	clk	Clock Input	*Double-click to*	**sdram...**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0400 0000	0x07ff_ffff
	wire	Conduit	**sdram_wire**			
	sdram_pll	ALTPLL Intel FPGA IP				
	inclk_interface	Clock Input	*Double-click to*	**clk_0**		
	inclk_interface_reset	Reset Input	*Double-click to*	[inclk_i...		
	pll_slave	Avalon Memory Mapped ...	*Double-click to*	[inclk_i...	0x0800 50e0	0x0800_50ef
	c0	Clock Output	*Double-click to*	sdram_...		
	c1	Clock Output	**sdram_clk**	sdram_...		
	sysid_qsys_0	System ID Peripheral Inte...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	control_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50f8	0x0800_50ff
	usb_irq	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50b0	0x0800_50bf
	external_connection	Conduit	**usb_irq**			
	usb_gpx	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50a0	0x0800_50af
	external_connection	Conduit	**usb_gpx**			
	usb_rst	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5090	0x0800_509f
	external_connection	Conduit	**usb_rst**			
	keycode	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5080	0x0800_508f
	external_connection	Conduit	**keycode**			
	leds_pio	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5070	0x0800_507f
	external_connection	Conduit	**leds**			
	VGA_text_mode_contoller_0	VGA Text Mode Controller				
	RESET	Reset Input	*Double-click to*	[CLK_1]		
	VGA_port	Conduit	**vga_port**	[CLK_1]		
	CLK_1	Clock Input	*Double-click to*	**clk_0**		
	avl_mm_slave	Avalon Memory Mapped ...	*Double-click to*	[CLK_1]	0x0800 0000	0x0800_3fff
	jtag_uart_0	JTAG UART Intel FPGA IP				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	avalon_jtag_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5100	0x0800_5107
	irq	Interrupt Sender	*Double-click to*	[clk]		1
	hex_digits_pio	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50d0	0x0800_50df
	external_connection	Conduit	**hex_digits**			
	key	PIO (Parallel I/O) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 50c0	0x0800_50cf
	external_connection	Conduit	**key_external_c...**			
	timer_0	Interval Timer Intel FPGA...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5000	0x0800_503f
	irq	Interrupt Sender	*Double-click to*	[clk]		2
	spi_0	SPI (3 Wire Serial) Intel F...				
	clk	Clock Input	*Double-click to*	**clk_0**		
	reset	Reset Input	*Double-click to*	[clk]		
	spi_control_port	Avalon Memory Mapped ...	*Double-click to*	[clk]	0x0800 5040	0x0800_505f
	irq	Interrupt Sender	*Double-click to*	[clk]		3
	external	Conduit	**spi0**			

Description of Platform Designer Modules

Not all of the modules pictured above are used for Lab 7. The ones that are used are described below.

Clk_0 is a module that functions as the 50 MHz clock from the FPGA. This module outputs clk and clk_reset which are used as clock and reset inputs for most of the other modules

Nios2_gen2_0 is the 32 bit processor that is the centerpiece of the design. In this case, the version of NIOS II that is used is the NIOS II/e which is the resource optimized version of the processor. The processor sends a data output and instruction output to other modules.

SDRAM is the off-chip memory that is used in both labs. It has a size of 512 MBits and takes input from the data and instruction masters outputted by the NIOS II. An interesting thing to note about the sdram is that the input clock comes from the sdram_pll modules. This clock is phase shifted to allow the SDRAM to perform a data read or write in the correct window with respect to the master.

SDRAM_PLL creates the clock that is outputted to the sdram. This clock is delayed by 1ns to compensate for the difference between the master and slave clock phase.

Sys_id_qsys_0 checks for incompatibilities between the hardware and the software that are being connected. This is used as a preventative measure so that the FPGA is not overloaded and possibly ruined by the software that it is connected to.

Timer_0 is a 1-bit module that sends a signal every millisecond so that the NIOS II is able to keep track of how much time has passed.

Spi_0 is a module that allows for data transfer between master and slave and vice versa through a MISO and MOSI channel. This can be done simultaneously depending on the mode of the SPI. The SPI also has a slave select that is active low. This is the part where our ‘four function’ controlling reading and writing bits was implemented.

VGA_text_mode_contoller_0 is a user created IP that implements the AVALON bus. This module is also responsible for sending the RGB signals to the VGA output via VGA_port.

Design Resources and Statistics

Lab 7.1

LUT	35703
DSP	0
Memory (BRAM)	11264 bits
Flip-Flop	21698
Frequency	67.21 Mhz
Static-Power	97.24 mW
Dynamic Power	249.24 mW
Total Power	368.74 mW

Lab 7.2

LUT	4893
DSP	0
Memory (BRAM)	49664 bits
Flip-Flop	2722
Frequency	75.52 Mhz
Static-Power	96.55 mW
Dynamic Power	66.96 mW
Total Power	185.55 mW

Conclusion

By the end of this lab, we were able to create fully functioning monochrome and colored text displays. This lab in conjunction with lab 6 were very helpful in understanding how to create a VGA display output. This lab, much more than lab 6, solidified the hardware and software interaction concepts.

The font rom character writing will be very useful when it comes to drawing sprites for the final project. Instead of characters, the idea can be extended to create larger figures, as well as multiple pictures of the same figure to resemble movement or rotations.

While this lab wasn't any more ambiguous than other labs, this lab was slightly more difficult than previous labs because of the novelty as it was difficult to receive help as many CA's had difficulty explaining the lab. I don't believe this lab should be changed because looking back, the algorithm to determine which character was really not complicated and, the lab was incredibly helpful in understanding software and hardware interaction as well as VGA drawing. In addition, next year's CA's will have a much better idea on how to explain what needs to be done in the lab.