VGA Text Mode Controller with Avalon-MM Interface

Hongbo Zheng, Yuhao Yuan

Summer 2021

ECE 385

Section AB1

TA: Hanfei Wang

Introduction

The purpose of this lab is to create a simplified text mode graphics controller which is connected to the Avalon memory-mapped bus and then add it to the Nios II system in order to support the 30 rows × 80 columns text mode through the VGA output. The data is writing into the 601 32-bit registers through the Avalon bus. Each of the register in the first 600 registers stores 4 groups of character code and inverse bit. The information of the character drawn on the VGA screen comes from the Font_Rom with the character code as its address. Algorithm and register decoding are required to figure out which character to draw at the X and Y position provided by the VGA_controller. The color of the character drawn on the VGA screen is decided by the RGB values stored inside the 601st register (the control register) and the inverse bit which swaps the foreground color and the background color.

Written Description of Lab 6 System

Lab 6 System

The entire lab 6 system is mostly the Nios II system, some hexdrivers, some inverters, and some tri-state buffers (last three are unused). The Nios II system contains Nios II/e Processor, on-chip memory, SDRAM controller, PLL for SDRAM, System ID Peripheral, several PIOs (Parallel I/O), JTAG UART, Interval Timer, SPI, and VGA Text Mode Controller. In lab 6, only the Nios II/e CPU, the VGA Text Mode Controller, and the Avalon bus are used to draw characters on the VGA monitor. The Nios II/e Processor is the master component which controls multiple slave peripherals through the Avalon bus, and VGA Text Mode Controller is one of peripherals. The VGA Text Mode Controller has many interfaces and components in it. It is able to read data from the Avalon bus and write data into the Avalon bus. Moreover, it contains VRAM (600 32-bit registers) which stores the information of the characters that are going to display on the screen. In addition, it has a Control Register which holds the RGB values of the foreground color and background color. Furthermore, it has Font_Rom which contains the drawing of the characters. It also has the VGA_controller which provides the real-time drawing location on the screen. Overall, all the components are working together through some logics (algorithms) to draw the correct characters with the correct color on the VGA monitor.

VGA Text Mode Controller IP

The VGA Text Mode Controller IP has 4 main interfaces in it: Clock, Reset, Conduit, and Avalon Memory Mapped Slave. The clock input has CLK as input signal. The reset input has RESET as input signal. The avl_mm_slave (Avalon Memory Mapped Slave) has 10-bit AVL_ADDR (address), 4-bit AVL_BYTE_EN (byte enable), 1-bit AVL_CS (chip select), 1-bit AVL_READ (read), 1-bit AVL_WRITE (write), and 32-bit AVL_WRITEDATA (write data) as input signals. Moreover, it has 32-bit AVL_READDATA (read data) as output signal. The VGA_port (Conduit) has 4-bit red, 4-bit green, 4-bit blue, 1-bit hs (horizontal sync), and 1-bit vs (vertical sync) as output signals.

The VGA Text Mode Controller IP is used as a high-level block which contains VRAM (600 × 32-bit reigster), Control Register (1 × 32-bit), Font_Rom, VGA_controller, R/W and Byte Access Logic, Avalon-MM Slave, and logic to draw the text characters from the VRAM and Font_Rom. The VRAM contains 600 32-bit registers which are used to store the character address and the invert bit. The Control Register is the 601th register, and it stores the RGB value of the foreground color and the background color. The

Font_Rom stores the shape of character in 16-bit rows × 8-bit columns dimension. The VGA_Controller provides the DrawX and DrawY signals which indicate the current drawing position on the VGA monitor. The Avalon-MM Slave and the R/W Byte Access Logic allow the registers to read the data from the Avalon bus or write the data into the Avalon bus. The logics that are used to draw the text characters on the VGA screen are some complicated algorithms. Specifically, the binary value of the pixel located at the position which is indicated by the DrawX and DrawY signals from the VGA_controller needs to be found first. Then, the color of that pixel is determined by the binary value of that pixel, the inverse bit from the register, and the RGB values in the Control Register. If the binary value of the pixel is '1', the character is displayed with the foreground color, and the background is displayed with the background color. When the invert bit has a binary value of '1' (active high), the foreground color and the background color are swapped. Overall, the VGA Text Mode Controller IP controls what is going to be drawn on the VGA monitor.

Logic used to read and write VGA registers

```
always_ff @(posedge CLK) begin
   if(RESET) begin
      for(int i = 0; i < `NUM_REGS; i++) begin
         LOCAL_REG <= 32'h0;
      end
   end
   else if(AVL_WRITE && AVL_CS)
   begin
      case(AVL_BYTE_EN)
         4'b1111: LOCAL_REG[AVL_ADDR][31:0]  <= AVL_WRITEDATA[31:0];
         4'b1100: LOCAL_REG[AVL_ADDR][31:16] <= AVL_WRITEDATA[31:16];
         4'b0011: LOCAL_REG[AVL_ADDR][15:0]  <= AVL_WRITEDATA[15:0];
         4'b1000: LOCAL_REG[AVL_ADDR][31:24] <= AVL_WRITEDATA[31:24];
         4'b0100: LOCAL_REG[AVL_ADDR][23:16] <= AVL_WRITEDATA[23:16];
         4'b0010: LOCAL_REG[AVL_ADDR][15:8]  <= AVL_WRITEDATA[15:8];
         4'b0001: LOCAL_REG[AVL_ADDR][7:0]   <= AVL_WRITEDATA[7:0];
         default:;
      endcase
   end
   else if(AVL_READ && AVL_CS)
   begin
      AVL_READDATA[31:0] = LOCAL_REG[AVL_ADDR][31:0];
   end
end
```

Since the logics are used to read and write registers, they should be inside the always_ff @ (posedge CLK). When RESET is high, all 601 32-bit registers should be storing logic value '0'. When AVL_WRITE and AVL_CS are both high, the data from the specific byte(s) of Avalon bus is written into the registers with the AVL_ADDR as the address based on the 4-bit AVL_BYTE_EN. The 4-bit AVL_BYTE_EN and their corresponding write actions are shown in the table below. When AVL_READ and AVL_CS are both high, the AVL_READDATA reads the entire 32-bit data stored inside the register at the address AVL_ADDR.

AVL_BYTE_EN [3:0]	Write Action	SystemVerilog
1111	Write full 32-bit	[31:0]
1100	Write the two upper bytes	[31:16]
0011	Write the two lower bytes	[15:0]
1000	Write byte 3 only	[31:24]
0100	Write byte 2 only	[23:16]
0010	Write byte 1 only	[15:8]
0001	Write byte 0 only	[7:0]

Algorithm used to draw text characters from VRAM and Font_Rom

```
logic [4:0] Row_Location, Register_in_Row, Font_Rom_Row;
logic [6:0] Column_Location;
logic [1:0] Location_in_Register;
logic [9:0] Register_Index;
logic [2:0] Font_Rom_Column;
logic [6:0] Font_Index;
logic Invert;
```

```
always_comb
begin
   Row_Location = DrawY>>4;
   Column_Location = DrawX>>3;
   Register_in_Row = Column_Location>>2;
   Location_in_Register = Column_Location[1:0];
   Register_Index = 20*Row_Location+Register_in_Row;

   Font_Rom_Row = DrawY[3:0];
   Font_Rom_Column = DrawX[2:0];

   if(Location_in_Register == 1'b0)
   begin
      Font_Index = LOCAL_REG[Register_Index][6:0];
      Invert = LOCAL_REG[Register_Index][7];
   end
   else if(Location_in_Register == 1'b1)
   begin
      Font_Index = LOCAL_REG[Register_Index][14:8];
      Invert = LOCAL_REG[Register_Index][15];
   end
   else if(Location_in_Register == 2'h2)
   begin
      Font_Index = LOCAL_REG[Register_Index][22:16];
      Invert = LOCAL_REG[Register_Index][23];
   end
   else
   begin
      Font_Index = LOCAL_REG[Register_Index][30:24];
      Invert = LOCAL_REG[Register_Index][31];
   end
end
```

The location of the current drawing pixel is indicated by the DrawX and DrawY from the VGA_controller. The color of the current drawing pixel is determined by the Invert bit and the binary value of the pixel. Therefore, the binary value of the Invert bit and the pixel have to be found by using some algorithm in order to determine which RGB value to use.

First of all, the index of the register which is in charge of the current drawing pixel has to be calculated. The Row_Location of the character is found by DrawY/16 which is the same as right-shift DrawY by 4 bits. The Column_Location of the character is calculated by DrawX/8 which is the same as right-shift DrawX by 3 bits. Then the Register_in_Row is calculated by Column_Location/4 which is the same as right-shift Column_Location by 2 bits because every 4 characters are stored inside 1 register. With the above calculations, the Register_Index can be found by 20 × Row_Location + Register_in_Row (offset).

Secondly, the correct 8-bit (1-bit Inverse bit + 7-bit character CODE) is required because there are 4 characters inside 1 register. The Location_in_Register is calculated by Column_Location % 4 which is the same as taking the last 2 bits (remainder) of Column_Location. Since the order of drawing is reversed on the VGA monitor, the CODE of the character needs to access with the reversed order shown in the table below.

Bit	31	30-24	23	22-16	15	14-8	7	6-0
Function	IV3	CODE3	IV2	CODE2	IV1	CODE1	IV0	CODE0
Character Index	0		1		2		3	
Remainder of Column_Location % 4	0		1		2		3	
Drawing Order	3		2		1		0	

Then, Font_Rom_Row needs to be calculated because the output of Font_Rom is 8-bit data which is 1 line of the 16-bit rows × 8-bit columns font data. The 4-bit Font_Rom_Row is found by DrawY % 16 which is the same as taking the last 4 bits (remainder) of the DrawY. Therefore, the 11-bit address of the 8-bit output data in Font_Rom is the concatenation of the 7-bit CODE decoded from the register and the 4-bit Font_Rom_Row.

11-bit Address of 8-bit output data	Input [10:0] addr of Font_Rom	
{7-bit,4-bit} Concatenation	7-bit CODE decoded from the Register	4-bit Row Index from 4-bit Font_Rom_Row

Finally, the binary value of the pixel is needed to decide which RGB values to use from the Control Register. Since there are 8-bit of output data from the Font_Rom, the column index (Font_Rom_Column) of the pixel is required to get the correct 1-bit value of the pixel from the 8-bit output data. The Font_Rom_Column is calculated by DrawX % 8 which is the same as taking the last 3 bits (remainder) of the DrawX. In addition, each row out of the 16 rows of the character is drawn in reversed order, so the correct column index of the pixel is actually 7−Font_Rom_Column. Thus, the correct binary value of the pixel is found to be data[7−Font_Rom_Column].

Implementation of Inverse Color Bit

```
else if(Invert == 1'b0)
begin
   if(data[7-Font_Rom_Column] == 1'b1)
   begin
      red   <= LOCAL_REG[600][24:21];
      green <= LOCAL_REG[600][20:17];
      blue  <= LOCAL_REG[600][16:13];
   end
   else
   begin
      red   <= LOCAL_REG[600][12:9];
      green <= LOCAL_REG[600][ 8:5];
      blue  <= LOCAL_REG[600][ 4:1];
   end
end
else if(Invert == 1'b1)
begin
   if(data[7-Font_Rom_Column] == 1'b0)
   begin
      red   <= LOCAL_REG[600][24:21];
      green <= LOCAL_REG[600][20:17];
      blue  <= LOCAL_REG[600][16:13];
   end
   else
   begin
      red   <= LOCAL_REG[600][12:9];
      green <= LOCAL_REG[600][ 8:5];
      blue  <= LOCAL_REG[600][ 4:1];
   end
end
```

When the Invert bit from the register is logic ‘0’, it means that the pixel is drawn in foreground color if the pixel has a binary value of ‘1’, and the pixel is drawn in background color if the pixel has a binary value of ‘0’. However, when the Invert bit from the register is logic ‘1’, the cases are swapped. Specifically, the pixel is drawn in background color if the pixel has a binary value of ‘0’, and the pixel is drawn in foreground color if the pixel has a binary value of ‘1’.

Implementation of Control Register

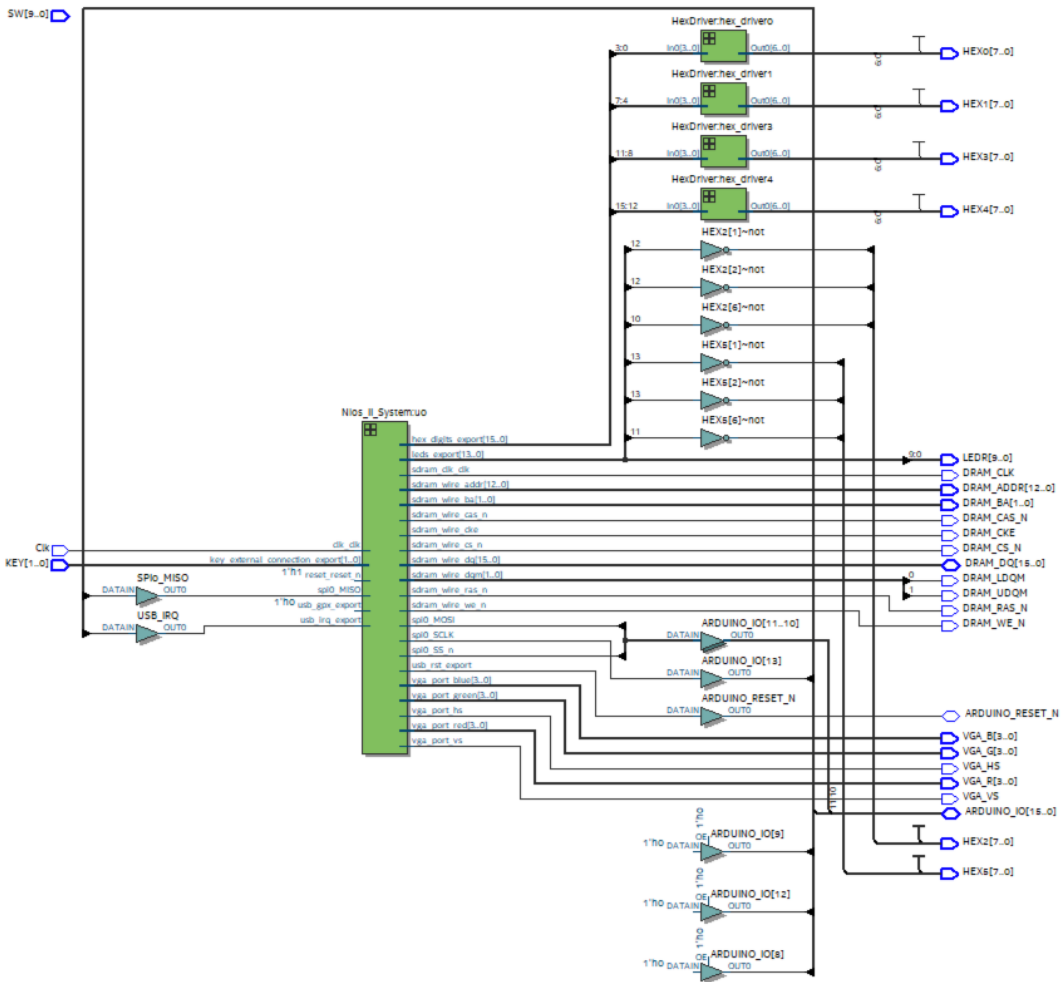
```
else if(Invert == 1'b0)
begin
   if(data[7-Font_Rom_Column] == 1'b1)
   begin
      red   <= LOCAL_REG[600][24:21];
      green <= LOCAL_REG[600][20:17];
      blue  <= LOCAL_REG[600][16:13];
   end
   else
   begin
      red   <= LOCAL_REG[600][12:9];
      green <= LOCAL_REG[600][ 8:5];
      blue  <= LOCAL_REG[600][ 4:1];
   end
end
else if(Invert == 1'b1)
begin
   if(data[7-Font_Rom_Column] == 1'b0)
   begin
      red   <= LOCAL_REG[600][24:21];
      green <= LOCAL_REG[600][20:17];
      blue  <= LOCAL_REG[600][16:13];
   end
   else
   begin
      red   <= LOCAL_REG[600][12:9];
      green <= LOCAL_REG[600][ 8:5];
      blue  <= LOCAL_REG[600][ 4:1];
   end
end
```

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

The Control Register is the 601st register with an index of 600. The [24:21] bits are the 4-bit Red of the foreground color. The [20:17] bits are the 4-bit Green of the foreground color. The [16:13] bits are the 4-bit Blue of the foreground color. The [12:9] bits are the 4-bit Red of the background color. The [8:5] bits are the 4-bit Green of the background color. The [4:1] bits are the 4-bit Blue of the background color. The outputs: 4-bit red, 4-bit green, and 4-bit blue, are assigned to FGD_R, FGD_G, FGD_B or BKG_R, BKG_G, BKG_B based on the Invert bit and the binary value of the pixel. The implementation is stated in the previous section.

Block Diagram and Module Description

High-Level Block Diagram of Lab 6



Module Description

Module: lab6.sv

Input: Clk, [1:0] KEY, [9:0] SW

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

Inout: [15:0] ARDUINO_IO, ARDUINO_RESET_N

Description: This component contains all the components connected with each other, such as the Nios II system, Hex Drivers, inverters, and tri-state buffers.

Purpose: The module is used as the top-level of the project which draws the characters on the VGA monitor.

High-Level Block Diagram of Nios II System

Module Description

Module: Nios_II_System.v

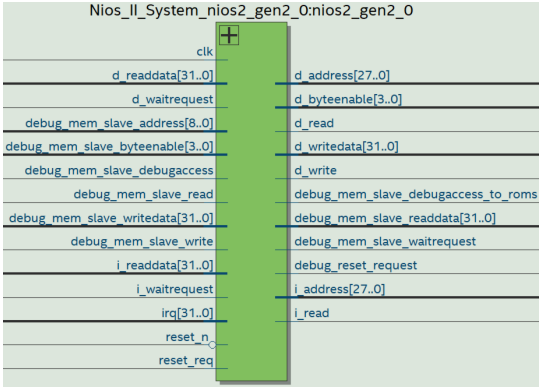
Input: clk_clk, [1:0] key_external_connection_export, reset_reset_n, spi0_MISO, usb_gpx_export, usb_irq_export

Output: [15:0] hex_digits_export, [13:0] leds_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wires_cas_n, sdram_wire_cke, sdram_wire_cs_n, [15:0] sdram_wire_dq, [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_green, vga_port_hs, [3:0] vga_port_red, vga_port_vs

Description: This component is the Nios II System which contains Nios II/e Processor, on-chip memory, SDRAM controller, PLL for SDRAM, System ID Peripheral, several PIOs (Parallel I/O), JTAG UART, Interval Timer, SPI, and VGA Text Mode Controller.Purpose: The module is used as the top-level of the project which draws the characters on the VGA monitor.

Purpose: This module is the high-level block of the Nios II System.

High-Level Block Diagram of Nios II CPU



Module Description

Module: Nios_II_System _nios2_gen2_0.sv

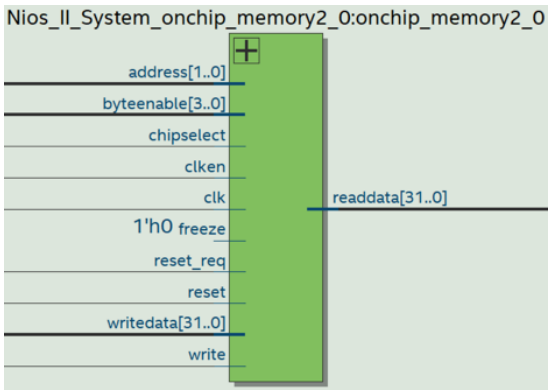
Input: clk, [31:0] d_readdata, d_waitrequest, [8:0] debug_mem_slave_address, [3:0] debug_mem_slave_byteenable, debug_mem_slave_debugaccess, debug_mem_slave_read, [31:0] debug_mem_slave_writedata, debug_mem_slave_write, [31:0] i_readdata, i_waitrequest, [31:0]irq, reset_n, reset_req

Output: [27:0] d_address, [3:0]d_byteenable, d_read, [31:0] d_writedata, d_write, debug_mem_slave_debugaccess_to_roms, [31:0]debug_mem_slave_readdata, debug_mem_slave_waitrequest, debug_reset_request, [27:0] i_address, i_read

Description: This component is the Nios II/e CPU.

Purpose: This module is used as the center processing unit of the project.

High-Level Block Diagram of Nios_II_System_onchip_memory2_0



Module Description

Module: Nios_II_System _onchip_memory2_0.sv

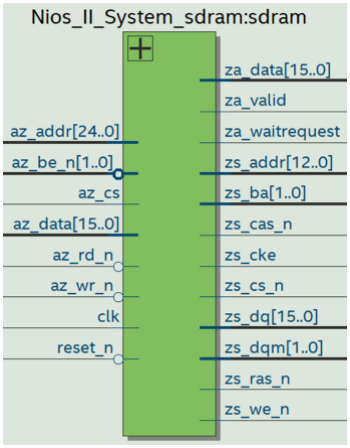
Input: [1:0] address, [3:0] byteenable, chipselect, clken, clk, freeze, reset_req, reset, [31:0] writedata, write

Output: [31:0] readdata

Description: This component is the Nios II on-chip memory.

Purpose: This module is used as on-chip memory of the Nios II System which is able to read and write faster than SDRAM because it is physically located closer to the Nios II CPU.

High-Level Block Diagram of Nios_II_System_sdram



Module Description

Module: Nios_II_System_sdram.sv

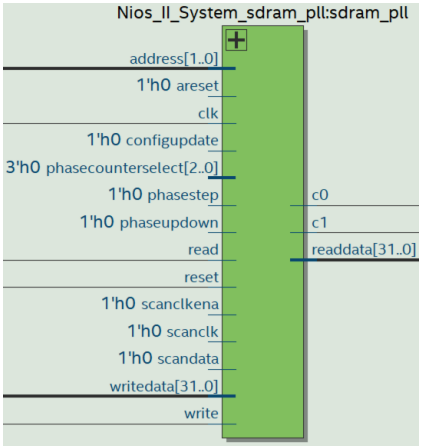
Input: [24:0] az_addr, [1:0] az_be_n, az_cs, [15:0] az_data, az_rd_n, az_wr_n, clk, reset_n

Output: [15:0] za_data, za_valid, za_waitrequest, [12:0] zs_addr, [1:0] zs_ba, zs_cas_n, zs_cke, zs_cs_n, [15:0] zs_dq, [1:0] zs_dqm, zs_ras_n, zs_we_n

Description: This component is the SDRAM of the Nios II System.

Purpose: This module is used as the memory component of the Nios II System. It has larger capacity than the on-chip memory, but it performs slower than the on-chip memory.

High-Level Block Diagram of Nios_II_System_sdram_pll



Module Description

Module: Nios_II_System_sdram_pll.sv

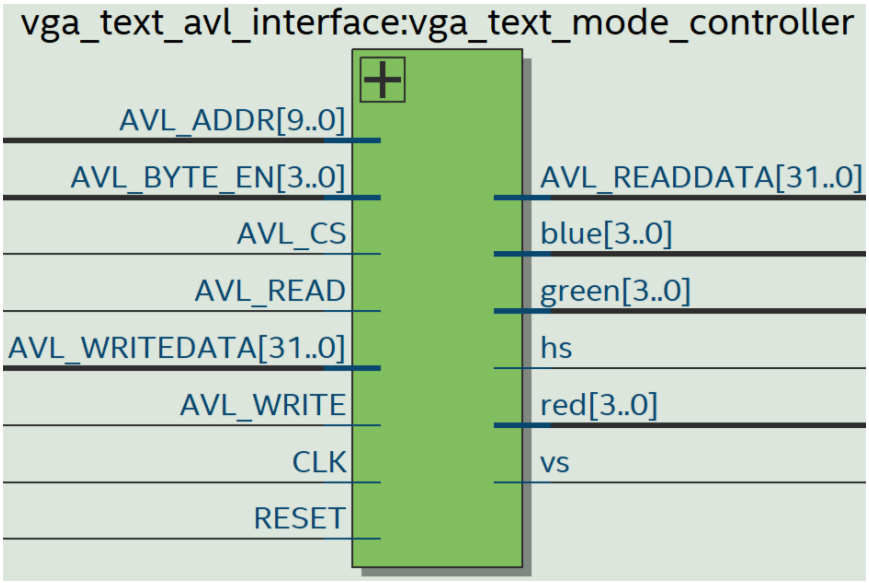
Input: [1:0] address, areset, clk, configupdate, [2:0] phasecounterselect, phasestep, phaseupdown, read, reset, scanclkena, scanclk, scandata, [31:0] writedata, write

Output: c0, c1, [31:0] readdata

Description: This component is clock component for the SDRAM.

Purpose: This module provides the clock signal for the SDRAM because the SDRAM requires precise timing.

High-Level Block Diagram of VGA Text Mode Controller



Module Description

Module: vga_text_avl_interface.sv

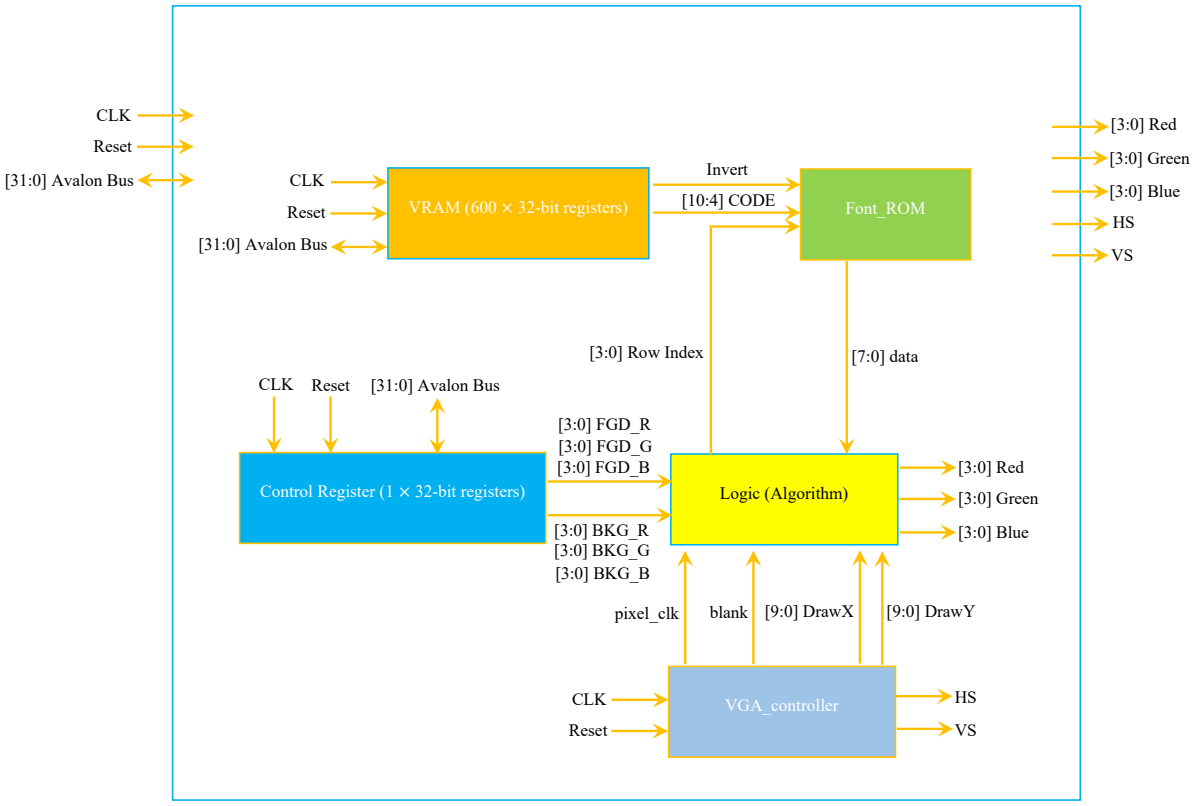
Input: [9:0] AVL_ADDR, [3:0] AVL_BYTE_EN, AVL_CS, AVL_READ, [31:0] AVL_WRITEDATA, AVL_WRITE, CLK, RESET

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

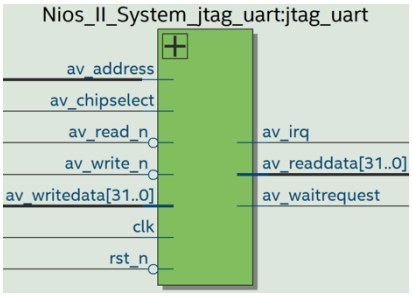
Description: This component is high-level block of the VGA Text Mode Controller which contains many interfaces and components. It has Avalon-MM Slave, R/W Byte Access Logic, VRAM (600×32-bit registers), Control Register (1 × 32-bit registers), Font_Rom, VGA_controller, and some logics (algorithms). The VGA Text Mode Controller is able to draw characters with specific colors on the VGA monitor.

Purpose: This module is the high-level block of VGA Text Mode Controller.

Block Diagram of VGA Text Mode Controller



High-Level Block Diagram of Nios_II_System_jtag_uart



Module Description

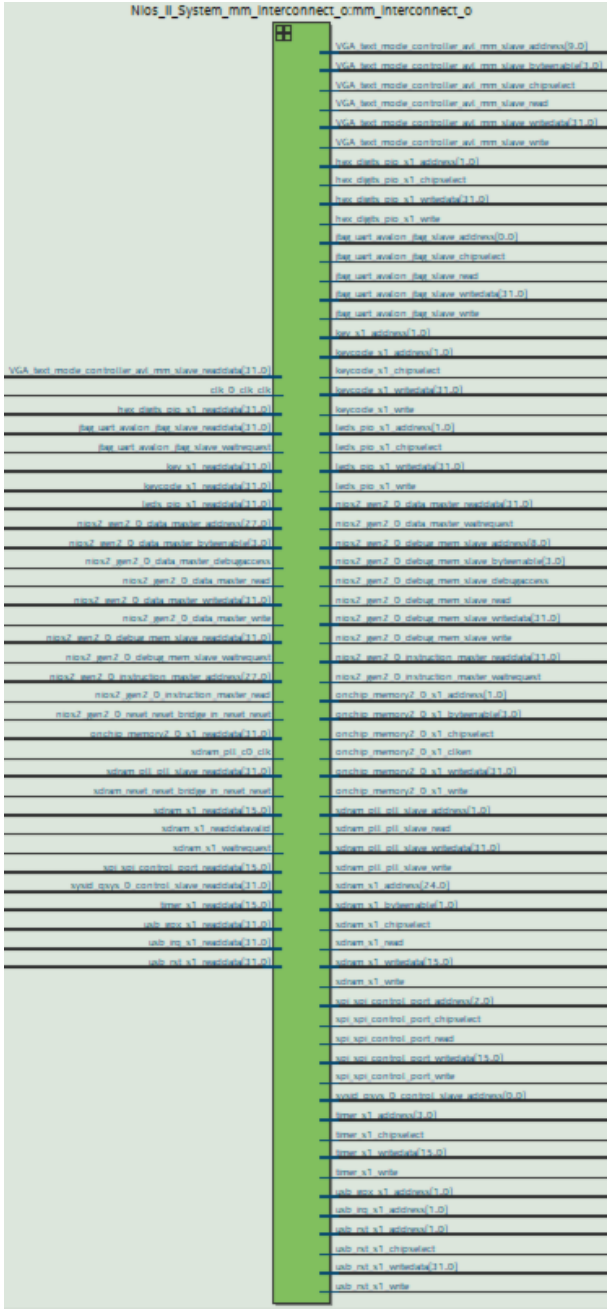
Module: altera_reset_controller.sv

Input: av_address, av_chipselect, av_read_n, av_write_n, [31:0] av_writedata, clk, rst_n

Output: av_irq, [31:0] av_readdata, av_waitrequest

Purpose: This component implements a method to communicate serial character streams between a host PC and an SOPC Builder system on FPGA.

High-Level Block Diagram of Nios_II_System_mm_interconnect_0



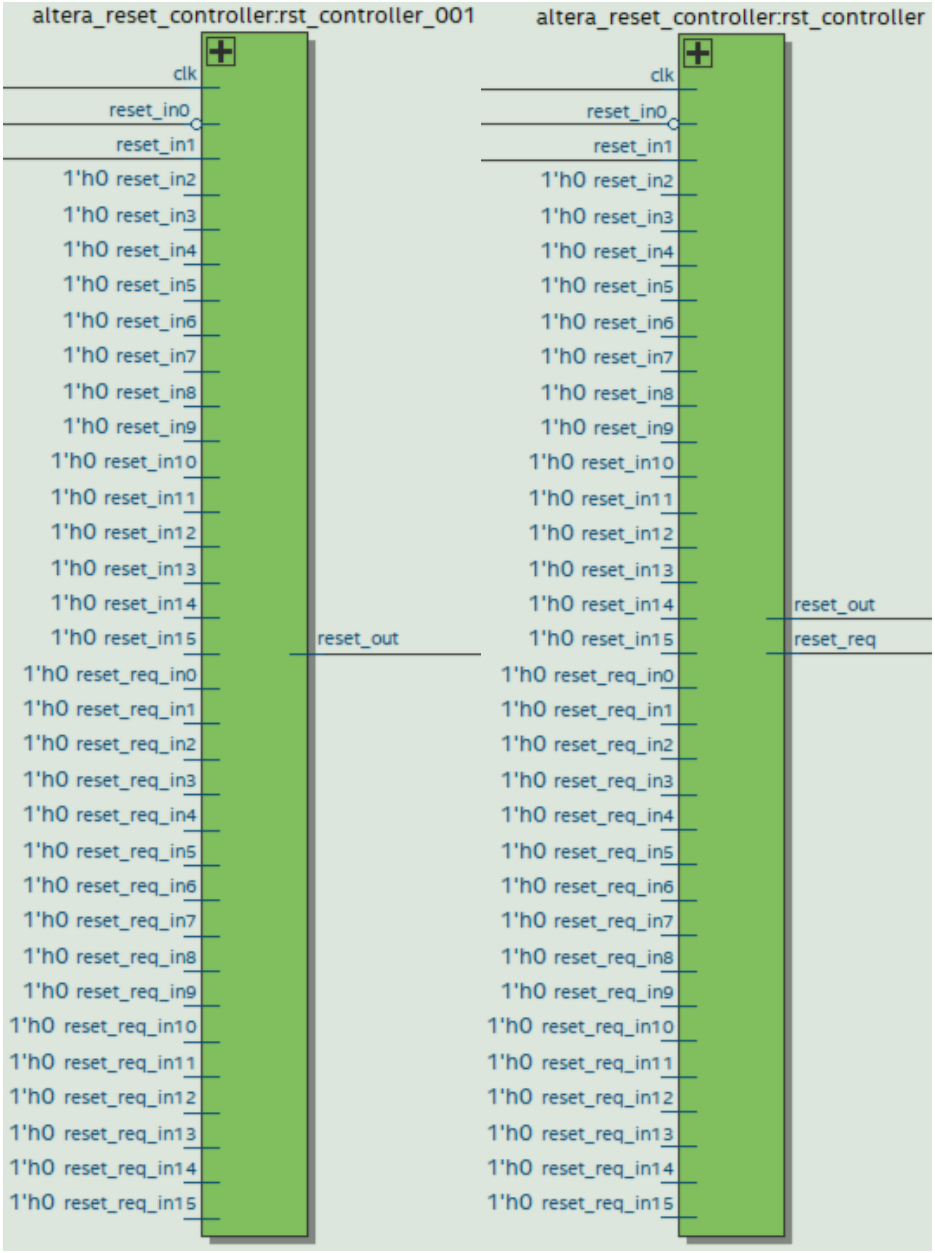
Module Description

Module: vga_text_avl_interface.sv

Description: This component is high-level block of the 32-bit Avalon bus which provides the data writing from the master component into the slave components or the data reading from the slave components to the master component.

Purpose: This module is the high-level block of the 32-bit Avalon data bus.

High-Level Block Diagram of altera_reset_controller



Module Description

Module: altera_reset_controller.sv

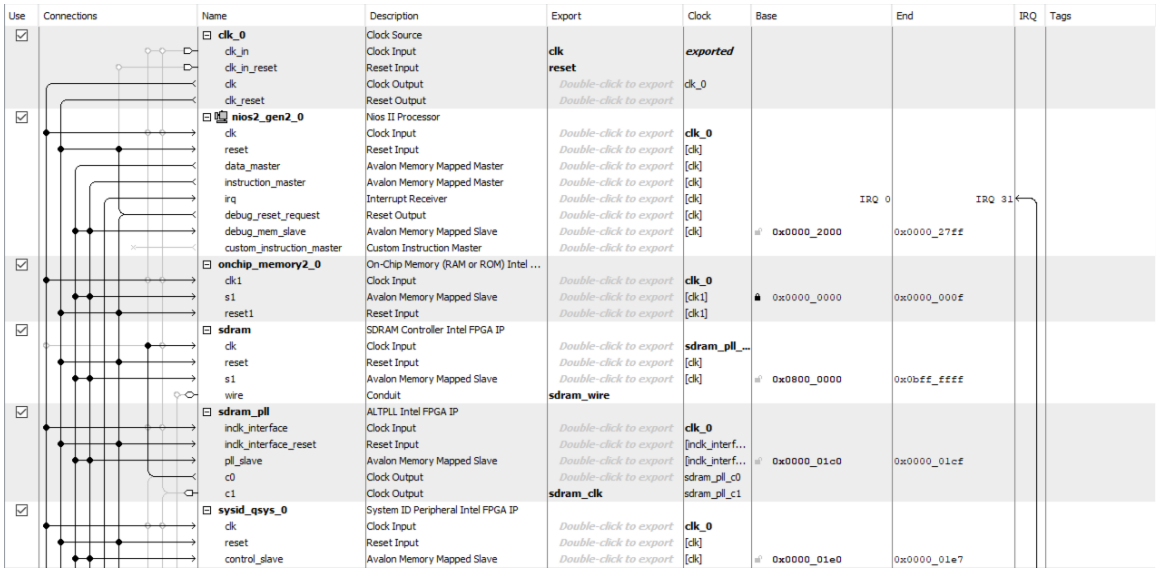
Input: clk, [15:0] reset_req_in

Output: reset_out, reset_req

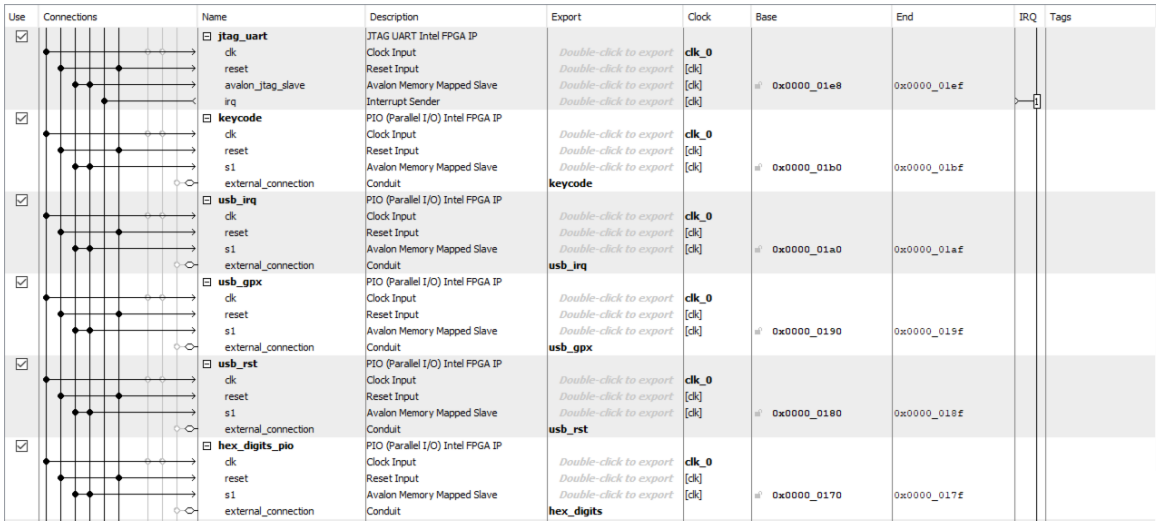
Description: This component is the reset controller of the Nios II system

Purpose: It is used to reset various component in the Nios II system.

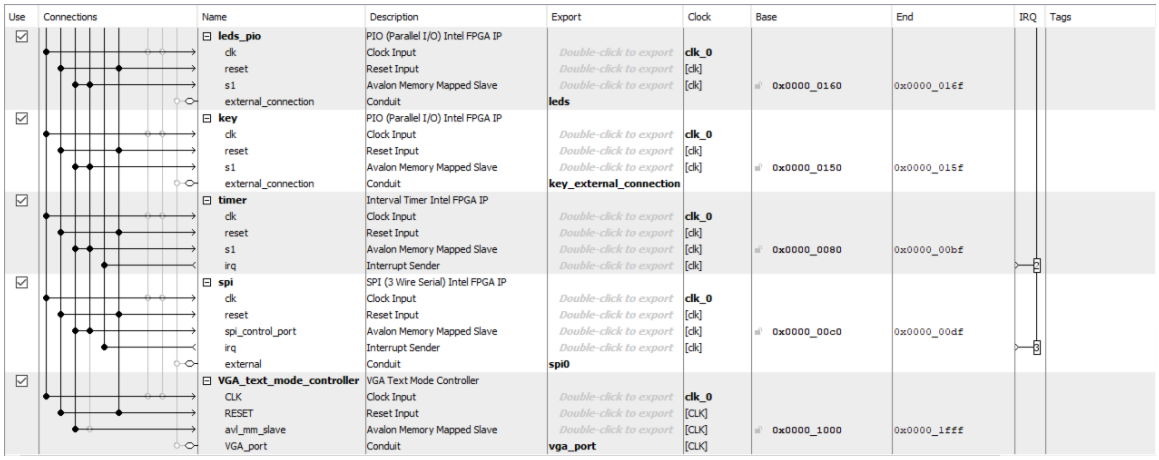
Platform Designer Module (Hardware Component)



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
☑		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	***exported***				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to export*	clk_0				
		clk_reset	Reset Output	*Double-click to export*					
☑		**nios2_gen2_0**	Nios II Processor						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]				
		irq	Interrupt Receiver	*Double-click to export*	[clk]			IRQ 0	IRQ 31
		debug_reset_request	Reset Output	*Double-click to export*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_2000	0x0000_27ff		
		custom_instruction_master	Custom Instruction Master	*Double-click to export*					
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM) Intel ...						
		clk1	Clock Input	*Double-click to export*	**clk_0**				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	0x0000_000f		
		reset1	Reset Input	*Double-click to export*	[clk1]				
☑		**sdram**	SDRAM Controller Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**sdram_pll_...**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_0000	0x0bff_ffff		
		wire	Conduit	**sdram_wire**					
☑		**sdram_pll**	ALTPLL Intel FPGA IP						
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**				
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_interf...				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interf...	0x0000_01c0	0x0000_01cf		
		c0	Clock Output	*Double-click to export*	sdram_pll_c0				
		c1	Clock Output	**sdram_clk**	sdram_pll_c1				
☑		**sysid_qsys_0**	System ID Peripheral Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01e0	0x0000_01e7		



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
☑		**jtag_uart**	JTAG UART Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01e8	0x0000_01ef		
		irq	Interrupt Sender	*Double-click to export*	[clk]			1	
☑		**keycode**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01b0	0x0000_01bf		
		external_connection	Conduit	**keycode**					
☑		**usb_irq**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01a0	0x0000_01af		
		external_connection	Conduit	**usb_irq**					
☑		**usb_gpx**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0190	0x0000_019f		
		external_connection	Conduit	**usb_gpx**					
☑		**usb_rst**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0180	0x0000_018f		
		external_connection	Conduit	**usb_rst**					
☑		**hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0170	0x0000_017f		
		external_connection	Conduit	**hex_digits**					



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
☑		**leds_pio**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0160	0x0000_016f		
		external_connection	Conduit	**leds**					
☑		**key**	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0150	0x0000_015f		
		external_connection	Conduit	**key_external_connection**					
☑		**timer**	Interval Timer Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_0080	0x0000_00bf		
		irq	Interrupt Sender	*Double-click to export*	[clk]			0	
☑		**spi**	SPI (3 Wire Serial) Intel FPGA IP						
		clk	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clk]				
		spi_control_port	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_00c0	0x0000_00df		
		irq	Interrupt Sender	*Double-click to export*	[clk]			2	
		external	Conduit	**spi0**					
☑		**VGA_text_mode_controller**	VGA Text Mode Controller						
		CLK	Clock Input	*Double-click to export*	**clk_0**				
		RESET	Reset Input	*Double-click to export*	[CLK]				
		avl_mm_slave	Avalon Memory Mapped Slave	*Double-click to export*	[CLK]	0x0000_1000	0x0000_1fff		
		VGA_port	Conduit	**vga_port**	[CLK]				

PIO Name: keycode

Inputs: clk, reset, [15:0] data_master

Output: keycode

Address: x01b0 – x01bf

Purpose: It takes the data from the keyboard and send it to FPGA.

PIO Name: usb_irq

Inputs: clk, reset, [15:0] data_master

Output: usb_irq

Address: x01a0 – x01af

Purpose: It is used to send interrupt request of the keyboard.

PIO Name: usb_gpx

Inputs: clk, reset, [15:0] data_master

Output: usb_gpx

Address: x0190 – x019f

Purpose: It is used to send input data of the keyboard.

PIO Name: usb_rst

Inputs: clk, reset, [15:0] data_master

Output: usb_gpx

Address: x0180 – x018f

Purpose: It is used to reset output of the usb.

PIO Name: hex_digits_pio

Inputs: clk, reset, [15:0] data_master

Output: hex_digits

Address: x0170 – x017f

Purpose: It is used as the hexadecimal output.

PIO Name: leds_pio

Inputs: clk, reset, [15:0] data_master

Output: leds

Address: x0160 – x016f

Purpose: It is used as the output of the LEDs.

PIO Name: key

Inputs: clk, reset, [15:0] data_master

Output: key_external_connection

Address: x0150 – x015f

Purpose: It is used as the input of key information from the keyboard.

Bugs encountered and corrective measures taken

The first part of the lab is creating a Platform Designer component which is called the "VGA text mode controller". This portion is pretty straightforward because the "Introduction to the Avalon-MM Interface and VGA Graphics" provide very detailed instructions on how to create the component and add it to the Nios II system. The most difficult and time-consuming part is to figure out the algorithm of drawing the correct color on each pixel. Initially, the algorithm (find the correct register), the decode of registers and the color outputs are all put inside the always_ff @ (posedge pixel_clk). However, the VGA monitor displays all the characters with glitches, some of them are not fully printed out, and some of them are not printed very clearly. Moreover, there are also several colored pixels displayed randomly on the background of the screen. Therefore, the problem is suspected to be the wire delay of the circuit. Since the results of algorithm and register decoding need to be available as fast as possible after the value DrawX and DrawY are changed regardless of the pixel_clk, it is believed that these two portions should be done outside the always_ff @ (posedge pixel_clk). After moving the algorithm and the decode of the registers inside to an always_comb, the VGA monitor starts to display the characters without the glitches. All the characters are displayed clearly and separately from each other, and there are no more colored pixels displayed randomly on the background of the screen.

Unfortunately, the content of the sentences displayed on the VGA monitor is incorrect. At first, it is suspected to be the mistakes of the algorithm and the register decoding because the address of font_rom is wrong which causes the problem of printing out the incorrect character on screen. However, it turns out that they are correct after doing some tests on paper. Then, we check the offset between the correct character and the character displayed on the screen (the wrong character). It seems like all the characters do not have the same offset, and they are off by different amount of value (same characters are off by same amount of value). This bug is not fixed before demo time, so there are some methods that we could try if we have more time. The first one is to hardware VRAM to some know value and see if the program works correctly. The second one is trying to get the content stored inside the register to check if it is storing the correct 4

characters. The third one is to check if the translation from byte addresses to word addresses (4-byte) is correct.

Post-Lab

Design Resources and Statistics

	Text Mode Graphics Controller
LUT	34,784
DSP	None
Memory (BRAM)	55,296
Flip-Flop	21,801
Frqeuency	67.11MHz
Static Power	97.30mW
Dynamic Power	225.98mW
Total Power	345.34mW

Conclusion

The functionality of this lab is to configure a basic SOC design with a CPU and memory, and then create a simplified text mode graphics controller IP core which is connected to the Avalon memory-mapped bus to support 30 rows * 80 columns text mode through the VGA output. The “Introduction to the Avalon-MM Interface and VGA Graphics” is very helpful for creating the text mode graphics controller IP core because it has very detailed instruction on how to set up the IP core in Platform Designer. In addition, the “Introduction to VGA Sprite Drawing” gives a very simple example on how to draw a single character in font_rom on the VGA monitor with specific RGB colors. It helps us understand how the VGA drawing works in SystemVerilog. Overall, this lab further expands the abilities of the FPGA and gets us to know how to draw complex things on the VGA monitor. Therefore, it enables us to create more complicated and more advanced project, such as Tetris and Pac-Man, on the FPGA with the VGA monitor and USB keyboard.