**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**

Text

Description automatically generated

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**

Text

Description automatically generated

Text

Description automatically generated

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 7Font\_Rom\_Column. Thus, the correct binary value of the pixel is found to be data[7Font\_Rom\_Column].

**Implementation of Inverse Color Bit**

A picture containing text, receipt

Description automatically generated

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**

A picture containing text, receipt

Description automatically generated

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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**

Diagram, schematic

Description automatically generated

**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**

A picture containing timeline

Description automatically generated

**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**

Table, timeline

Description automatically generated with medium confidence

**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**

A picture containing timeline

Description automatically generated

**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**

A picture containing chart

Description automatically generated

**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**

A picture containing graphical user interface

Description automatically generated

**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**

A picture containing timeline

Description automatically generated

**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 (60032-bit registers), Control Register (132-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**

Diagram

Description automatically generated

**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**

**A close-up of a graph

Description automatically generated with medium confidence**

**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**

A picture containing timeline

Description automatically generatedTimeline

Description automatically generated

**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)**

Graphical user interface, application, table

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface, application

Description automatically generated

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.