ECE385

Fall 2021

Experiment #7

VGA Text Mode Controller with Avalon-MM Interface

Haozhe Si

haozhes3

Section ABD

1. Introduction

a) Basic Operation of VGA Interface

The VGA interface we designed would communicate with the CPU and the VGA port. It will read/write the data for VRAM according to the instruction of CPU; fetch the font and the color from the FONT_ROM according to the text mode VRAM; and output the RGB color of the pixel the VGA controller is scanning. In summary, the basic operation of VGA interface is to map color to VGA port according to the software input.

b) Improvement from Lab 6.2

In lab 6.2, we implemented the color mapper on System Verilog. To be specific, in last lab, the ball position is generated by the Ball module. Meanwhile, the Color Mapper module will decide whether a pixel is the background pixel or ball, and assign the corresponding color to it. Therefore, all the color assigning and mapping procedures are done on the hardware. In contrast, in this lab, the image we want to display on screen is assigned by the software. The software will write data to VRAM and the VGA interface will assign color according to the VRAM. Therefore, in this lab, color assigning is done by software, which is the improvement build on Lab 6.2.

2. System Description

a) Lab 7 System

The core hardware components in this lab including the NIOS-II/e Processor and the VGA Text Mode Controller designed by us. We also have a JTAG_UART module to get and print out the software outputs. Finally, we have the a serious of PIO modules and modules for USB ports and SPI, clock source module, the SDRAM controller module and the system ID Peripheral. For the System Verilog modules, we have the design file for the VGA Text Mode Controller, which implements the function of the VGA Text Mode Interface. In addition, we have the VGA Controller module to scan the pixels on the screen; the Font ROM that saves the fonts of the supported characters; and the VRAM module used in the second part of the lab that initialized the on-chip memory.

b) VGA Text Mode Controller

The VGA Text Mode Controller mainly performs two tasks: handling the communication between the CPU and the VRAM, and mapping color to the screen according to the VRAM and the FONT ROM. To achieve the first task, we implemented an Avalon Memory Mapped Slave on hardware level, which includes address, write data, read data, read enable, write enable, byte enable and chip select. In the first lab, we set the read data wait for one clock cycle after the read enable and write data will not wait after write enable. To

achieve the second task, we implemented the VGA port that will read data from VRAM, fetch font from FONT ROM and output the RGB value of pixels, horizontal synchronization, and vertical synchronization signals. The details are discussed below.

c) Read/Write of VGA Registers

In Lab 7.1, we implemented the VRAM using VGA registers. There are 601 32-bit registers for the VRAM, 600 for characters and 1 for controlling foreground and background colors. The registers are accessed by the AVL_ADDR signal which specifies the address. To read from the VGA register to CPU, both AVL_CS and AVL_READ need to be set high. Since reading from registers needs time, we will read the AVL_READDATA signal one clock cycle later then the AVL_CS and AVL_READ signals being set, and thus we put the data assignment inside a sequential logic block. To read from the CPU to VGA register, both AVL_CS and AVL_WRITE need to be set high, and to synchronize the data with clock, we also put it inside the sequential logic block. Notice that the AVL_BYTE_EN signal will specify which Bytes to write inside the 32-bit register, and we are going to write the corresponding bits from AVL_WRITEDATA to the VGA registers.

d) Draw Characters

To draw characters, we first instantiated the VGA controller. The VGA controller will generate the electron beam and output the coordinate of pixel is it scanning on (*DrawX*, *DrawY*), along with the *hs*, *vs*, *blank* and *pixel_clk*. To draw characters on the screen, the VGA Text Mode Interface will first calculate the VGA register from the pixel coordination. To be specific, the we will decide whether the current pixel is background or foreground by:

$$character\_x\_offset = DrawX\,[2:0]$$
$$character\_y\_offset = DrawY\,[3:0]$$
$$sprite\_x\_offset = DrawX >> 3$$
$$sprite\_y\_offset = DrawY >> 4$$
$$sprite\_index = sprite\_y\_offset * 80 + sprite\_x\_offset$$
$$vram\_offset = sprite\_index\,[1:0]$$
$$vram\_addr = sprite\_index >> 2$$

With the *vram_addr* and *vram_offset* we can then find the corresponding *font_code* from VRAM. Notice that the characters in the VGA registers are specified in little endian, and the characters need to be accessed accordingly. With *font_code*, we then can calculate the *font_addr* and inverse bit by:

$$font\_addr = fontcode\,[6:0] * 16 + character\_y\_offset$$
$$inv = fontcode\,[7]$$

By feeding the *font_addr* into FONT_ROM module, we can get the *font_data* for the character font at the specified vertical location. Finally, *font_data* [3'b111 - *character_x_offset*] will be the pixel value of the current pixel location, where 1 means it is the foreground and 0 mean background.

According to the pixel value and the color in the control register, we can decide the RGB output for the pixel.

e) Control Register and Inverse Bit

We can fetch out the foreground RGB color and background RGB color directly from the control register, which is last register in the VGA registers. To decide whether a pixel should be assigned with background color or the foreground color, we can XOR the pixel value with the *inv* signal, and if the result is 1, the pixel needs to be assigned with foreground color and 0 means background color. To synchronize the color mapping with the screen monitor frequency, this step is implemented in the sequential logic synchronizing with the *pixel_clk*.

f) On-Chip Memory VRAM

We instantiated a two ports OCM to serve as the VRAM in lab 7.2. Since there are two ports, one port for the OCM is used for the communication between the CPU and the other is used in character fetching for the VGA color mapping. Our OCM has synchronized inputs and outputs for data and addresses, read and write enables and one byte enable for the AVL_WRITE from CPU to OCM.

g) Hardware Component Modification for Lab 7.2

Since the OCM have flip-flops for both input and output ports, the output signal for read AVL_READDATA needs to wait for two clock cycles to have the valid output after the AVL_READ and AVL_ADDR signals are set. Therefore, we need to increase the wait time for the read signal in Component Designer IP to 2 clock cycles.

h) Sprite Drawing Algorithm Modification for Lab 7.2

Since every character can have its own foreground and background color set, we now need 1200 32-bit VRAM, where each 32-bit memory contains the character and color information for two sprites. Therefore, the new calculation for the indices become:

$$sprite\_x\_offset = DrawX >> 3$$
$$sprite\_y\_offset = DrawY >> 4$$
$$sprite\_index = sprite\_y\_offset * 80 + sprite\_x\_offset$$
$$delayed\_character\_x\_offset <= DrawX\,[2:0]$$
$$delayed\_character\_y\_offset <= DrawY\,[3:0]$$
$$delayed\_vram\_offset <= sprite\_index\,[1:0]$$
$$vram\_addr <= sprite\_index >> 1$$
$$character\_x\_offset <= delayed\_character\_x\_offset$$
$$character\_y\_offset <= delayed\_character\_y\_offset$$
$$vram\_offset <= delayed\_vram\_offset$$

Notice that the calculations for the indices accessing the VRAM are all

implemented in sequential logic and are delayed for one more pixel clock cycle. This is because accessing the VRAM implemented by the OCM is delayed by the input and output flip-flops as discussed before.

i) Support Multicolored Text for Lab 7.2

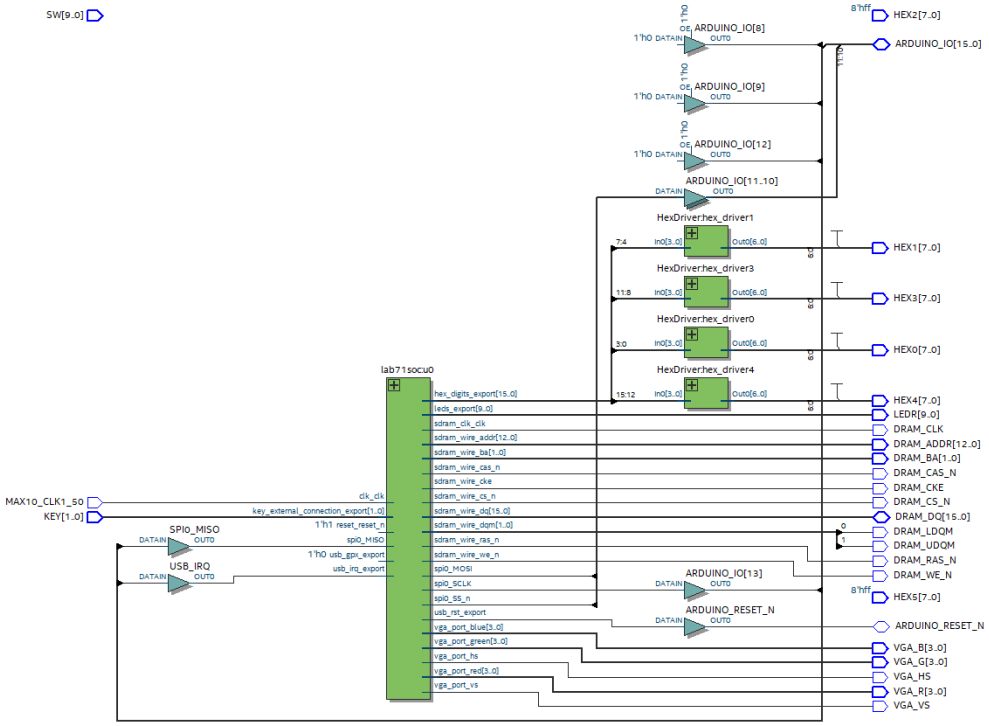
Given the *vram_offset* and the *vram_addr*, we can fetch the *font_code* and *font_color* from the VRAM. Since each 32-bit memory now contain two characters and their color information, we will select the corresponding bytes from the readout data of the OCM accordingly. With the *font_code*, we can calculate the pixel value and the *inv* as before, and we can decode the foreground and background color index from the *font_color* signal, which serves as the control register in Lab 7.1.

j) Support Color Palette for Lab 7.2

To implement the color palette, we instantiated 8 registers to save the 16 colors. The address of the color palette registers starts from 0x800, therefore, when performing data writing, we will first check the address, if the MSB is 0, then the data will be read from/write to the VRAM, while if the MSB is 1, the data will be read from/write to the palette registers. To access the color palette, we will first decode the *font_color*, fetching out the color index for the background and foreground colors, and assigning the RGB value for the foreground and background according to the palette. Then we can decide the output color for the pixel given the pixel value and *inv* as before.

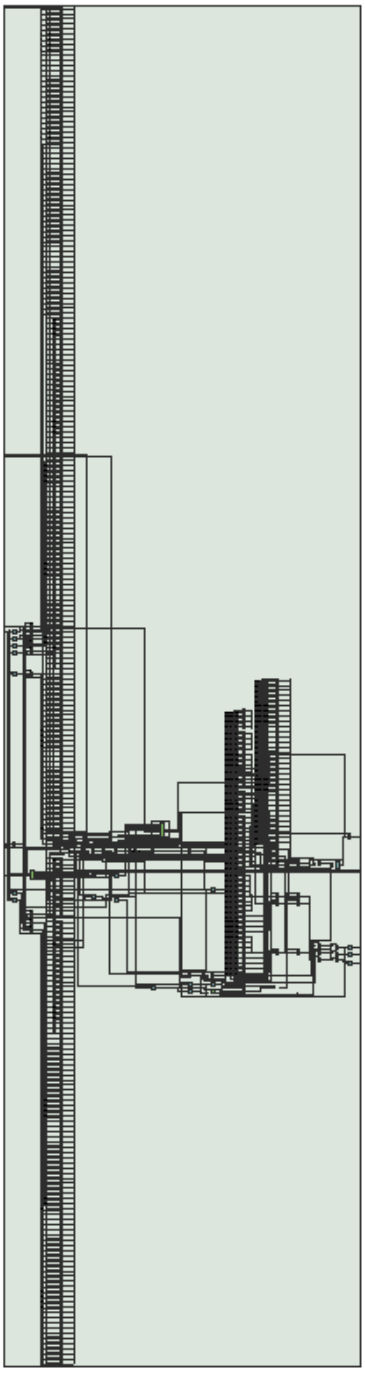
3. Block Diagram

a) Top Level



b) Lab 7.1 VGA Text Mode Controller

c) Lab 7.2 VGA Text Mode Controller



4. Module Description

a) Common Modules

Module: Lab7.sv
Inputs: [1:0] KEY, [7:0] SW, MAX10_CLK1_50
Outputs: [7:0] LEDR, [12:0] DRAM_ADDR, DRAM_CKE, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CS_N, DRAM_LDQM, DRAM_UDQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
I**nout:** [15:0] DRAM_DQ

Description: This module instantiated the module for NIOS-II system.
Purpose: This is the top-level module for interfacing with the NIOS-II system module. It connects the IOs and DRAM on the MAX10 board to the NIOS-II system module.

Module: Lab7soc.sv
Components: clk_0, nios2_gen2_0, VGA_text_mode_controller, leds_pio, sdram, sdram_pll, sysid_qsys_0, keycode, hex_digits_pio, spi_0, jtag_uart_0, timer_0, usb_irq, usb_gpx, usb_rst, key
Description: This module instantiated the components used in NIOS-II system. Among all the components, the VGA_text_mode_controller which specifies the hardware design is the most import module we used, and will take the System Verilog we wrote as the description for its operation.
Purpose: This is the System Verilog code for the SOC we defined in the QsYs. The components are connected according to the code we have here.

Module: VGA_controller
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, sync, blank, [9:0] DrawX, [9:0] DrawY
Description: This module will first divide the *Clk* speed in half and create the *pixel_clk*. Then module is then synchronized with the *pixel_clk*. The VGA_controller module will scan the pixel array of the monitor. Every time the it finishing scanning the length of a row, it will set the *hs* signal, and every time it finishing all pixels, it will set the *vs* signal. Since the actual scanning area is larger than the screen area, when the controller scans the area outside the monitor, it will set the *blank* signal. The module will also output the coordination of the pixels it is scanning at the current clock cycle.
Purpose: The purpose of this module is to decide the location of the pixel it is scanning. It also outputs the synchronization signals of the VGA hardware. This information will be later used to determine the color of the pixels.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: The module will output the code for seven-segment display depending on the input 4-bit data.
Purpose: This module is used to translate the binary data into the code for hex display on the board.

Module: vga_text_avl_interface
Inputs: [3:0] AVL_BYTE_EN, [11:0] AVL_ADDR, [31:0] AVL_WRITEDATA, AVL_READ, AVL_WRITE, AVL_CS
Outputs: [31:0] AVL_READDATA, [7:0] Red, [7:0] Green, [7:0] Blue, hs, vs
Description: This module instantiated the VRAM, VGA controller and Font

ROM. In lab 7.2, it also instantiated the palette. The implemented algorithms are discussed in previous sections.
Purpose: This module implemented the read/write interface between the CPU and the VRAM, fetch character from FONT ROM and map color from control registers/palettes to the VGA ports.

Module: VRAM
Inputs: [31:0] DATAWRITE, [10:0] readout_addr, [10:0] AVL_ADDR, [3:0] AVL_BYTE_EN, AVL_WREN, AVL_REN, clock, pixel_clk
Outputs: [31:0] DATAREAD, [31:0] readout_data
Description: This module instantiated the OCM used by the lab 7.2.
Purpose: This is the wrapper for the OCM.

Module: font_rom
Inputs: [10:0] addr
Outputs: [7:0] data
Description: This module instantiated the ROM for the 128 characters, each characters take 16*8-bit memories. The 8-bit data can be accessed by the input address.
Purpose: This module saves the font information for all the characters.

5. Design Analysis

1) Lab 7.1

LUT	32467
DSP	0
Memory (BRAM)	11264
Flip-Flop	3857
Frequency	128.27MHz
Static Power	96.18mW
Dynamic Power	0.53mW
Total Power	106.03mW

2) Lab 7.2

LUT	4703
DSP	0
Memory (BRAM)	49664
Flip-Flop	649
Frequency	120.13MHz
Static Power	96.18mW
Dynamic Power	0.95mW
Total Power	106.45mW

6. Conclusion

a) Functionality

The first part of the lab implements the functionality of enabling VGA text mode. We designed the VGA text mode controller and make the software can print characters onto screens. For the second part of the lab, we further moved the VRAM from the registers to the OCM and enables the color palette to have each character the ability to control their foreground and background color independently.

b) Potential Extension

This lab inspired me how to print tiles onto screens. By designing sprites and color palettes for the game elements, we can print the colorful game graphics on the screen. This can be useful for us since we are going to design an Atari game for final project, who consists of relativly simple game graphics.

c) Comment on Lab Manual

Everything is good.