ECE 385
Lab 9
VGA Text Mode Controller with Avalon-MM Interface

Li, Ziying
Kang, Xingjian

May 3, 2024

1 Introduction

In this lab, we used platform designer and Nios ii system in Quartus to build a program that we can print text with different colors on the VGA screen. We modified the color_mapper from lab8 to be able to output

2 Written Description of Lab 9 System

2.1 Week 1 (Monochrome Text Display)

2.1.1 i. Written Description of the entire Lab 9 system

The system used characters within ASCII range and we add color to the font. Then we used VGA display port to show these colorful letters on the screen.

2.1.2 ii. Describe at a high level your VGA Text Mode controller IP

The VGA Text Mode controller uses a top level SV module vga_text_avl_interface.
This module gets RAM inputs from Aavalon bus and outputs VGA display signals back to Avalon bus.
This module contains modules vga_controller, font_rom, and color_mapper.

2.1.3 iii. Describe the logic used to read and write your VGA registers

Write Logic:
Clock Trigger: The write operation is triggered on the positive edge of the CLK signal.
Write Condition: The write occurs when both AVL_WRITE and AVL_CS (Chip Select) signals are high. Byte Enable: The AVL_BYTE_EN signal is a 4-bit vector, where each bit represents the enable status for a byte within the 32-bit word. If AVL_BYTE_EN[0] is high, the lower 8 bits of the LOCAL_REG at the address AVL_ADDR are updated with the corresponding 8 bits from AVL_WRITEDATA. Similarly, if AVL_BYTE_EN[1], AVL_BYTE_EN[2], or AVL_BYTE_EN[3] are high, the next 8-bit chunks of LOCAL_REG are updated.
Data Write: The LOCAL_REG array is updated with the data from AVL_WRITEDATA based on the byte enable conditions.
Read Logic:
Clock Trigger: The read operation is also triggered on the positive edge of the CLK signal.
Read Condition: The read occurs when both AVL_READ and AVL_CS signals are high.
Data Output: The entire 32-bit word from LOCAL_REG at the address AVL_ADDR is copied to the AVL_READDATA output.

2.1.4 iv. Describe the algorithm used to draw the text characters from the VRAM and font ROM (specifically, describe the equations required to generate the correct addresses to index into the VRAM as well as the font ROM).

step1: Addressing the VRAM to Retrieve Character Information:

step1.1: Calculating VRAM Character Address (vram_addr_chara):

Row position (Row) is calculated by dividing the draw Y coordinate (DrawY) by 16 (assuming 16 rows of characters per display line). Column position (Col) is calculated by dividing the draw X coordinate (DrawX) by 8 (assuming 8 characters per display column). The VRAM character address is then calculated as Row * 80 + Col (assuming 80 columns of characters in VRAM).

step1.2: Determining Character ID (letter_id):

The character ID within the word at the VRAM character address is calculated by taking the modulo 4 of the VRAM character address (vram_addr_chara % 4). This is because each word in VRAM holds 4 characters (32 bits divided by 8 bits per character).

step1.3: Extracting ASCII Code and Inverse Flag from VRAM:

An array temp is populated with the bit indices of the 7 ASCII code bits and the inverse flag bit for the character identified by letter_id. The ASCII code is extracted from VRAM by indexing into the word at vram_addr_word using the bit indices from temp. The inverse flag (inverse) is extracted from the most significant bit of the character's byte in VRAM.

step2: Addressing the Font ROM to Retrieve Character Pixel Data:

step2.1: Calculating Font ROM Address (addr):

The font ROM address is calculated by multiplying the ASCII code by 16 (assuming 16 bytes of pixel data per character in the font ROM) and adding the remainder of DrawY divided by 16. This gives the offset within the character's pixel data for the current scanline.

step2.2: Get foreground and background color

Then we can retrieve foreground and background color values from a control register (CTRL_REG).

2.1.5 v. Describe your implementation of the inverse color bit, as well as the implementation of the control register.

The most significant bit of a character address (note that a word contains 4 characters) is used for inverse bit. We also reserve the last element of the LOCAL_REG for control register while the rest of the LOCAL_REG is used for VRAM.

2.2 Week 2 (Color Text Display)

2.2.1 Describe the hardware changes you had to make to support the use of multi-color text.

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

We use a dual-ported memory, and it has two independent access ports, one port for AVL's reading and writing, and port B for VGA screening's reading only.

2. Corresponding modifications to the Platform Designer IP (e.g. Part Editor).

We increase the number of accessible registers 32-bit registers to 1200, to support the addition of per-character color attributes.

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

VRAM word Address Calculation:

The VRAM address calculation has been updated to reflect the new indexing scheme. We have two characters in on word now, so vram_addr_word now is calculated by the formula vram_addr_chara/2. Also, the formula for letter_id is updated as: vram_addr_chara%2;

VRAM Data Extraction:

The code now conditionally extracts the character data (DrawChar), foreground color index (DrawFGD_IDX), and background color index (DrawBKG_IDX) based on the letter_id. If letter_id is 0, it reads from the upper word; otherwise, it reads from the lower word.

Color Lookup:

The color lookup logic has been updated to handle the new color index format. The code now conditionally selects either the upper or lower half of the 13-bit palette register (PALETTE_REG) based on the least significant bit of the color index.

4. Additional modifications necessary to support multicolored text.

We do not have other additional modifications to support multicoloer text.

5. Additional hardware/code to draw paletted colors

Initialization on Reset:

We added a reset condition that sets PALETTE_REG to its default value of all zeros when RESET is true. This ensures that the palette register is properly initialized when the system is reset.

Write Operation with Byte Enable:

The write operation is now conditional on AVL_WRITE and AVL_CS being true, as well as the address bit AVL_ADDR[11] being set. The write operation now supports byte-wise writes. The byte enable signal AVL_BYTE_EN determines which byte of AVL_WRITEDATA is written to the corresponding byte of PALETTE_REG based on the AVL_ADDR[2:0] address. If the byte enable is not set for a particular byte, the existing value in PALETTE_REG is retained.

Read Operation: The read operation is now conditional on AVL_READ and AVL_CS being true, as well as the address bit AVL_ADDR[11] being set. When a read operation is performed, the value stored at PALETTE_REG[AVL_ADDR[2:0]] is copied to the REG_READDATA register.

Multiplexing Read Data: We added an always_comb block to multiplex the AVL_READDATA output. If AVL_ADDR[11] is set, AVL_READDATA is assigned the value from REG_READDATA (which contains the value read from PALETTE_REG). If AVL_ADDR[11] is not set, AVL_READDATA is assigned the value from ram_out_a, presumably from another memory or register that is not shown in the provided code.

3 Block Diagram

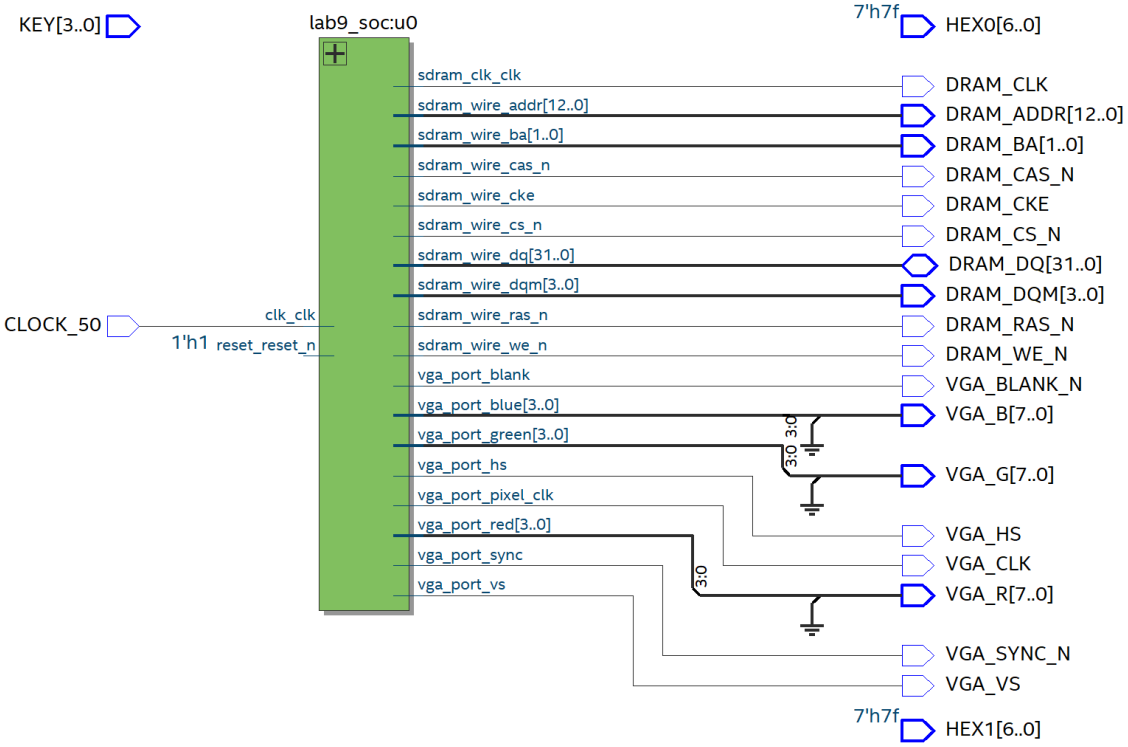

Figure 1: top level

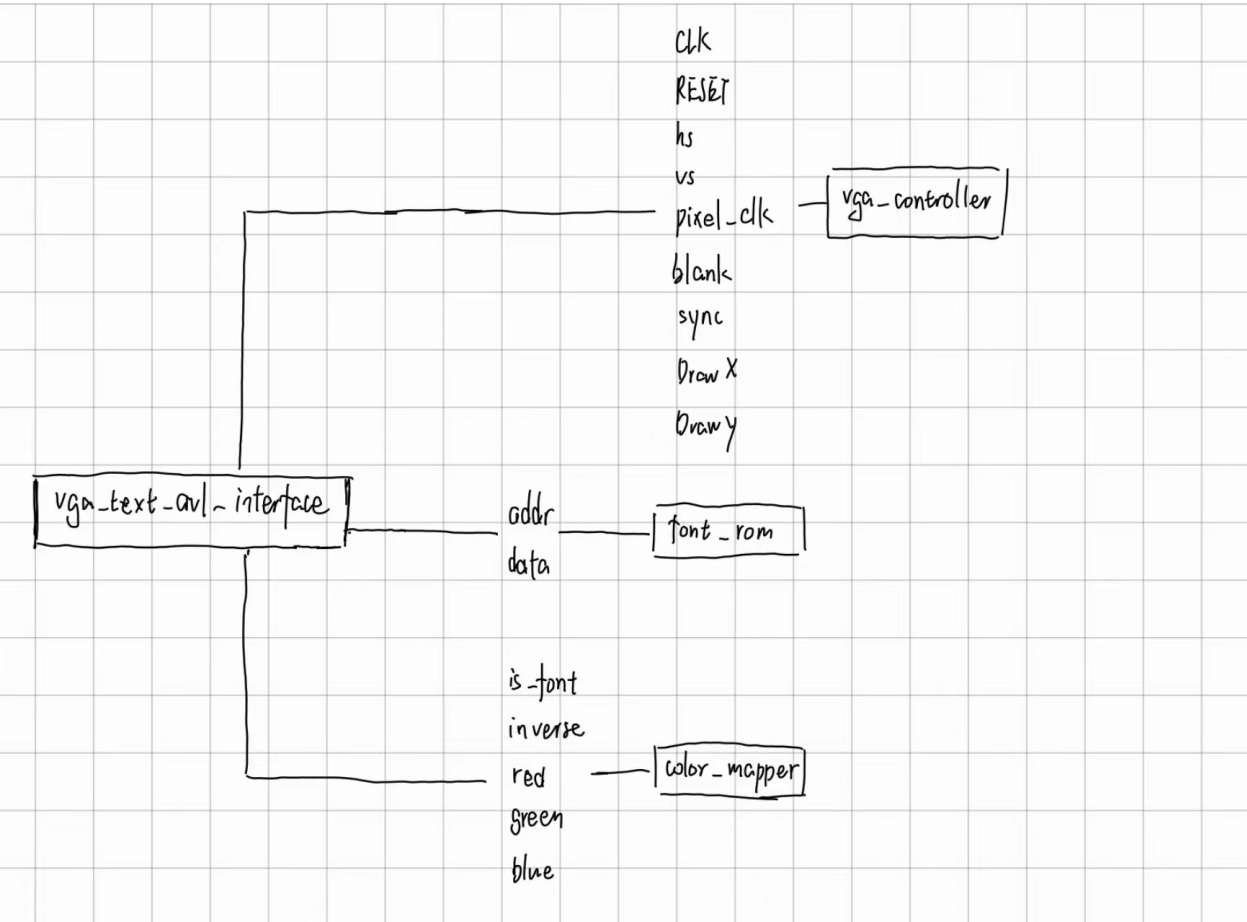

Figure 2: Common part

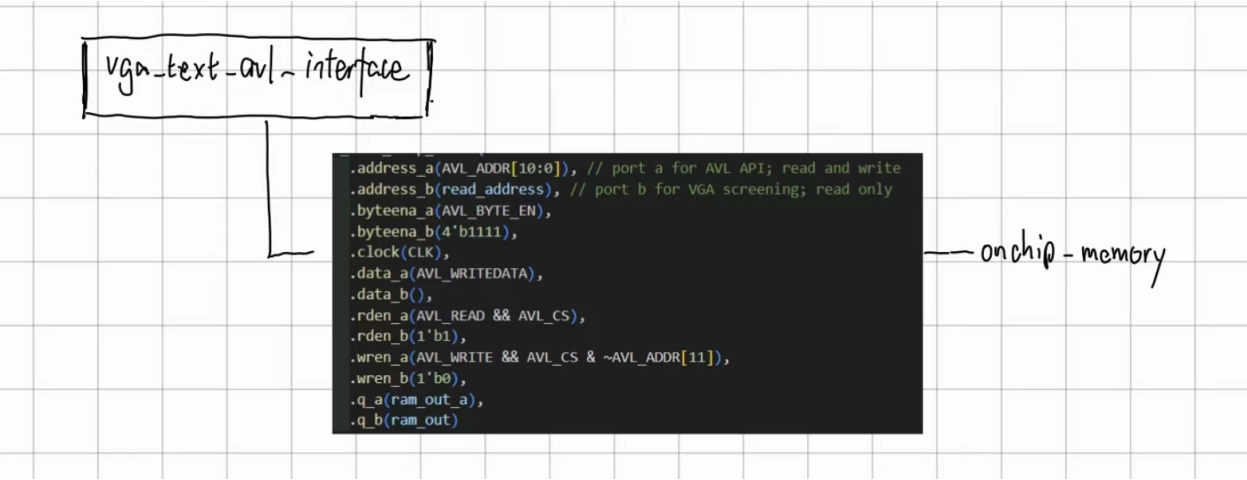

Figure 3: Week2 added

4 Module Descriptions

a. A guide on how to do this was shown in the Lab 7 report outline. Do not forget to describe the Platform Designer generated file for your Nios II system! When describing the generated file, you should describe the PIO blocks added beyond those just needed to make the NIOS system run (i.e. the ones needed to communicate with the USB chip and other components). The Platform Designer view of the Nios II system is helpful here.

(1) **Module:** vga_text_avl_interface

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

Outputs: logic [31:0] AVL_READDATA, logic [3:0] red, green, blue, logic hs, vs, logic sync, blank, pixel_clk

Description: top level module

Purpose: perform as interconnection and do basic logic

(2) **Module:** OCM_2

Inputs: [10:0] address_a, [10:0] address_b, [3:0] byteena_a, [3:0] byteena_b, clock, [31:0] data_a, [31:0] data_b, rden_a, rden_b, wren_a, wren_b,

Outputs: [31:0] q_a, [31:0] q_b

Description: Port A performs identically to the AVL memory interface, while Port B is read-only and responds to internal requests for VGA screening.

Purpose: Serves as a on-chip memory.

(3) **Module:** color_mapper

Inputs: is_font, inverse, FGD[11:0], BKG[11:0], DrawX[9:0], DrawY[9:0]

Outputs: VGA_R[3:0], VGA_G[3:0], VGA_B[3:0],

Description: This module takes in various inputs such as the pixel coordinates (DrawX, DrawY), a flag indicating if the pixel is part of a font (is_font), an inverse flag (inverse), and the foreground (FGD) and background (BKG) colors. Based on the is_font and inverse flags, it decides whether to output the foreground or background color for the RGB channels of the VGA. The RGB components are extracted from the respective color inputs (FGD or BKG) and assigned to the output ports VGA_R, VGA_G, and VGA_B.

Purpose: The purpose of this module is to decide which color (foreground or background) should be output to the VGA for each pixel based on the provided inputs.

(4) **Module:** font_rom

Inputs: [10:0] addr,

Outputs: [7:0] data

Description: read-only memory for font

Purpose: Convert ASCII code to 16*8 pixels display.

(5) **Module:** vga_controller

Inputs: Clk, Reset

Outputs: logic hs, vs, pixel_clk, blank, sync,

[9:0] DrawX, DrawY

Description: vga display port signal control

Purpose: The purpose of this module is to generate the timing signals required for VGA display. It produces horizontal and vertical sync pulses, a pixel clock, blanking interval indicators, and horizontal and vertical coordinates for each pixel.

5 Document the Design Resources and Statistics from the lab manual.

Each week's design should have different design statistics, and you should briefly discuss the difference between using on-chip memory for VRAM and registers. Which design is more efficient, what are the tradeoffs?

5.1 week 1

LUT	31678
DSP	0
Memory (BRAM)	55,296bits
Flip-Flop	21,423
Frequency	138.39 MHz
Static Power	105.27 mW
Dynamic Power	0.79 mW
Total Power	166.95 mW

5.2 week 2

LUT	4250
DSP	0
Memory (BRAM)	156,672bits
Flip-Flop	2459
Frequency	118.15 MHz
Static Power	102.06 mW
Dynamic Power	0.82 mW
Total Power	163.77 mW

6 Conclusion

6.0.1 Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.

In this lab, we design a simpler text mode graphics controller that connects to the Avalon memory-mapped bus and provides 80-column text mode via the VGA output. In the first week, we construct a reduced version of the monochromatic graphics adapter that simply supports black and white text. In the second week, we redesign and extend the graphics controller from week 1 to use on-chip memory, resulting in a bigger video memory (VRAM) space.

6.0.2 What are some potential extensions of this design, what did you learn in this lab that might be useful for your Final Project?

We learn how to use on-chip memory for final project. We also learn how to calculate the proper address to draw the correct character.

6.0.3 Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.

Overall, we think this lab's materials are great.