ECE 385

Spring 2024

Experiment # 9

Lab9 : VGA Text Mode Controller with Avalon-MM Interface

Name: Jie Wang, Shitian Yang

Student ID: 3200112404, 3200112415

Prof. Chushan Li,
Prof. Zuofu Cheng
ZJU-UIUC Institute
April 26, 2024, Friday D-225
TA: Jiebang Xia
Demo Point: 10/10

1. Introduction

a. Operation of the VGA Interface and Our Design Goal

The VGA (Video Graphics Array) interface is a standard for managing the display output to monitors. In Lab 9, the operation of the VGA interface involves creating a simplified text mode graphics controller that connects to the Avalon memory-mapped bus, supporting an 80-column text mode through VGA output. This design is similar to the functionality of early monochrome graphics adapters, allowing for the display of text characters in a designated format across the VGA screen. The main goal is to demonstrate how digital design can manipulate hardware to display text using VGA, integrating design elements like a font ROM and a video memory (VRAM) to map and display characters.

b. Improvement Based on Lab 8

Building upon Lab 8, which introduced basic VGA output, Lab 9 extends these capabilities to include a text mode controller. As we can generate VGA signal and draw basic lines, we need to support text rendering through a memory-mapped interface. This lab introduces new elements such as a font ROM for character data and the integration of text into the VRAM, which together facilitate the controlled display of text characters. The design moves from simply manipulating pixels to complex character displays, providing foundational skills for more advanced graphical tasks in final project. With lab9, we can easily realize a RPG game.

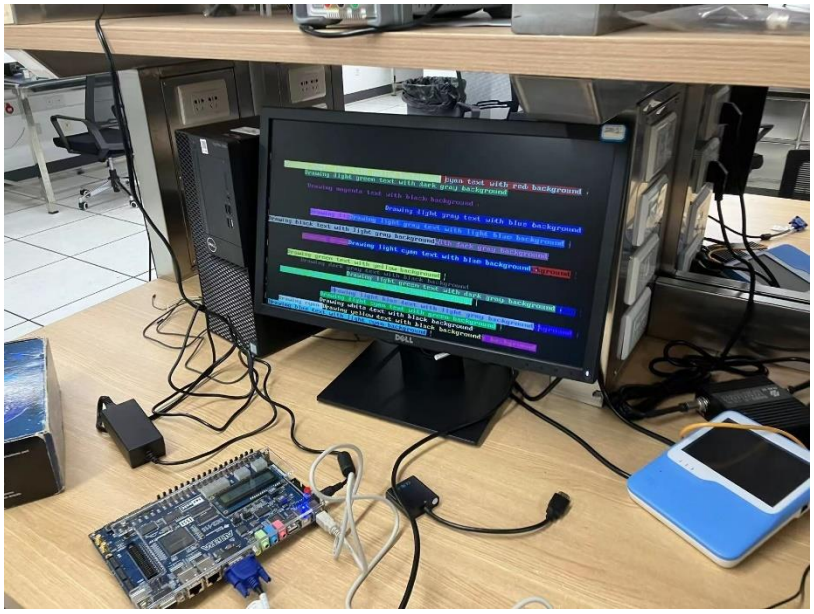

Fig-1: Random text display functionality

2. Description of Lab 9 System

a. Week 1: Monochrome Text Display

i. Written Description of the Entire Lab 9 System

In Week 1, we developed a simplified text mode VGA controller, interfaced with the Avalon-MM bus, supporting an 80-column text display. This controller closely emulates the functionalities of the early IBM monochrome graphics adapter (MGA). The system manages a display matrix of 80 columns by 30 rows, translating to 2400 characters, each specified by 8 bits. These bits include 7 bits for glyph selection from a subset of IBM codepage 437 and an additional bit for color inversion. The controller utilizes VRAM (2.4kBytes) mapped directly to the Avalon bus, facilitating modification of display content via the Nios II CPU.

ii. High-Level Description of VGA Text Mode Controller IP

The VGA text mode controller IP developed for this lab is primarily responsible for translating character data stored in VRAM into signals that drive the VGA display. This includes fetching glyph data for each character from the **font_rom.sv** module, processing this data based on the screen's raster scan needs, and handling inverted color display through control bits.

iii. Logic Used to Read and Write VGA Registers

The controller interfaces with the Avalon-MM bus to handle read and write operations. The Avalon-MM slave port uses signals like **read**, **write**, **readdata**, **writedata**, **address**, **byteenable**, and **chipselect** to perform memory operations. Each memory operation addresses 32-bit wide registers that map the VRAM and a control register. The system's design ensures a read latency of one cycle and a write latency of zero cycles, with byte-level control over data transactions through the **byteenable** signal.

iv. Algorithm to Draw Text Characters from VRAM and Font ROM

Characters are drawn by calculating the VRAM addresses to fetch corresponding character data, which includes both the glyph code and the inversion flag. The address calculation for VRAM is indexed linearly (raster order) where each word can store four characters. For the **font_rom**, the glyph code is used to fetch the appropriate 8x16 pixel bitmap, and the corresponding bits are processed to generate the VGA output based on the current **DrawX** and **DrawY** coordinates.

v. Implementation of the Inverse Color Bit and the Control Register

The inverse color functionality is managed by a dedicated bit in each character's data byte within VRAM. This bit toggles the foreground and background colors of the glyph. The control register, mapped at a specific word within the memory space, allows for the setting of global foreground and background colors, affecting the entire display unless overridden by the inverse color functionality at the character level.

b. Week 2: Color Text Display

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

We removed all the registers storing the VRAM. Instead, we use dual-port access to allow simultaneous read and write operations. The dual port RAM has one read and one write port to read and write signals which can set the avl_read and avl_ write data to the required value with the byte enable single. This setup helped in managing the limited ports available on the chip effectively by allowing the VGA controller to read data for display at the same time.

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

We expanded AVL_ADDR from 10 bits addressability to 12 bits addressability. Rather than accommodating data for four characters within a single 32-bit value, only two characters were stored per 32-bit value. This adjustment necessitated twice the VRAM capacity, leading to an expansion of the AVL_ADDR's address range from 10 bits to 12 bits.

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

In week 2, we doubled the size of registers to 1200, and enabled the characters with different foreground color and background color. But the equations are almost the same as week 1.
num = DrawY[9:4] * 80 + DrawX[9:3]
Instead of pulling the word_data from local registers, we get word_data from RAM module. Similar to Week 1, we used char to calculate which character to display as well as print inverted colors or not.
sprite_addr = {CODEn, DrawY[3:0]}
If (sprite_data[3'b111 - DrawX[2:0]] ^ IVn)....

iv. Additional modifications necessary to support multicolored text

To support multicolored text, each character cell in the VRAM was enhanced to store not only the character data but also color attributes for the foreground and background colors. We increased the addressability of AVL_ADDR from 10 bits to 12 bits to store the additional information. A color palette system was implemented, utilizing FPGA registers to store the RGB values for different colors. Specifically, 8 FPGA registers were designated as color palette registers, each holding 32-bit values that represent the RGB values for two colors. The addresses ranging from 0x800 to 0x807 were allocated for the color palette, allowing for easy access and management of color data.

v. Additional hardware/code to draw palette colors

We set up 8 palette registers within the FPGA, ranging from addresses 0x800 to 0x807. Each 32-bit register stores the RGB values for two colors, with each color occupying 12 bits. In each 32-bit value stored for each character in the on-chip memory, which includes the indexes for the foreground and background colors. And colors are retrieved from the palette registers based on the index value. If the index value is even, the rightmost colors from the registers are accessed; if odd, the leftmost colors are accessed.

3. Block diagram

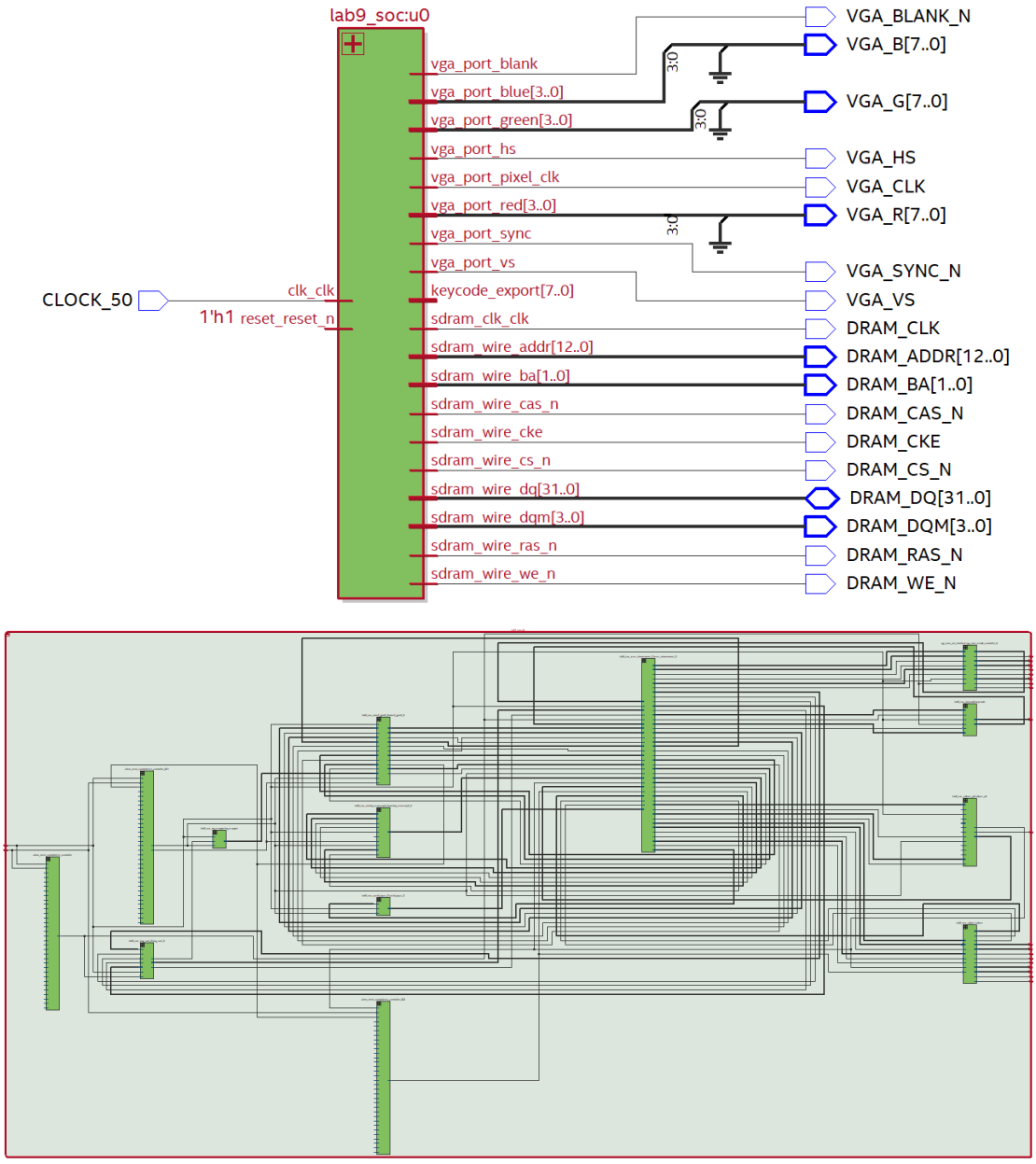

Fig-3: RTL Viewer of the Lab9 SV Module for week 1

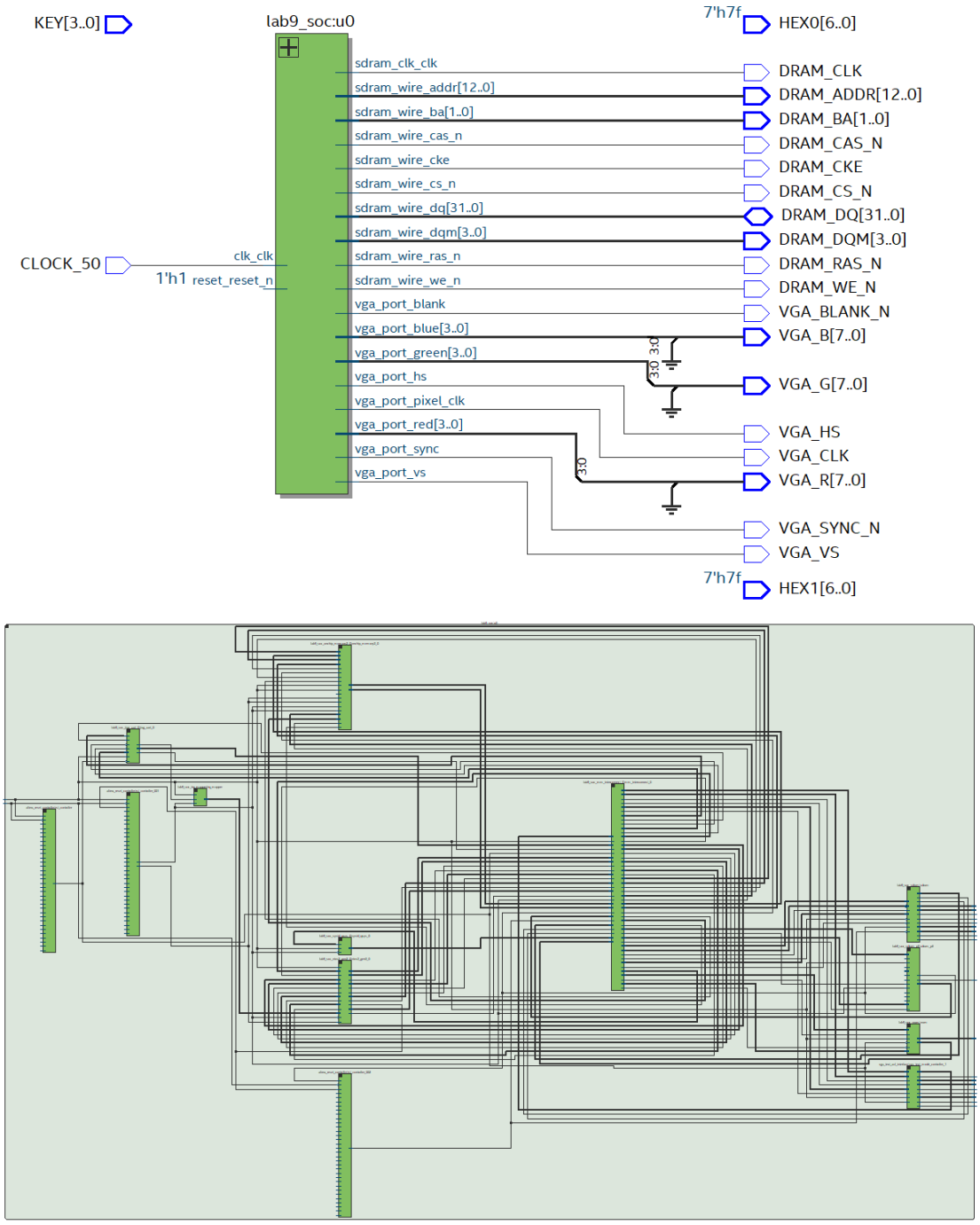

Fig-4: RTL Viewer of the Lab9 SV Module for week 2

4. Module Description

Please read **Appendix A.**

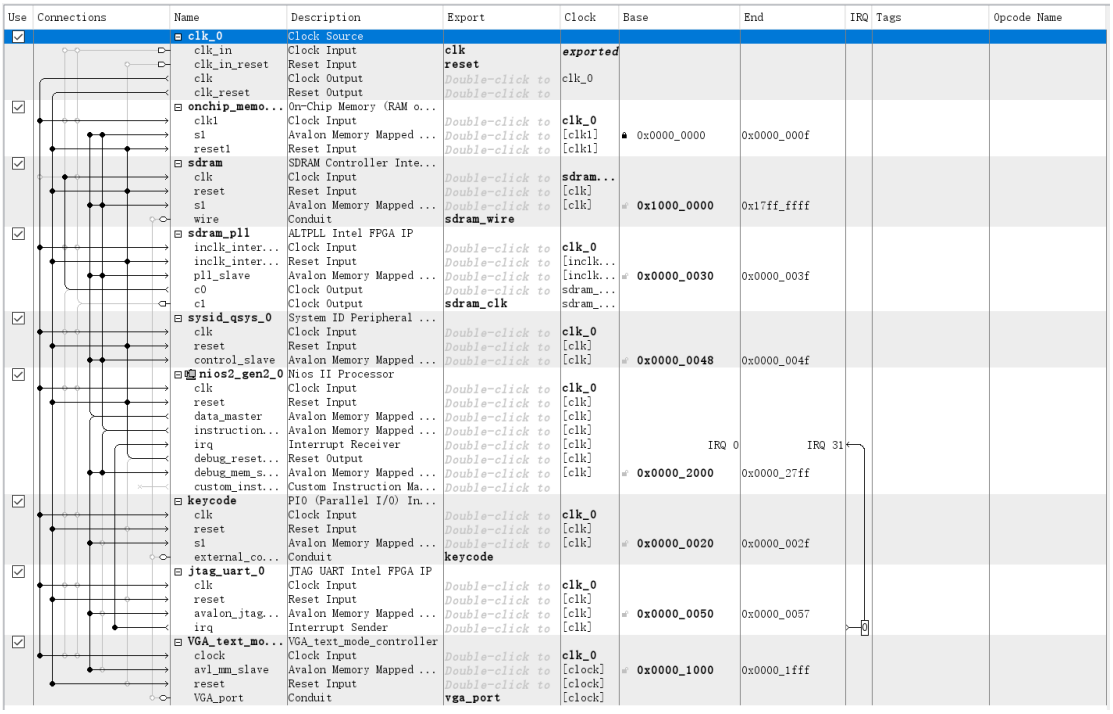

Fig-5: Our Connection in lab9_soc for week 1, adding a self-defined IP for text display

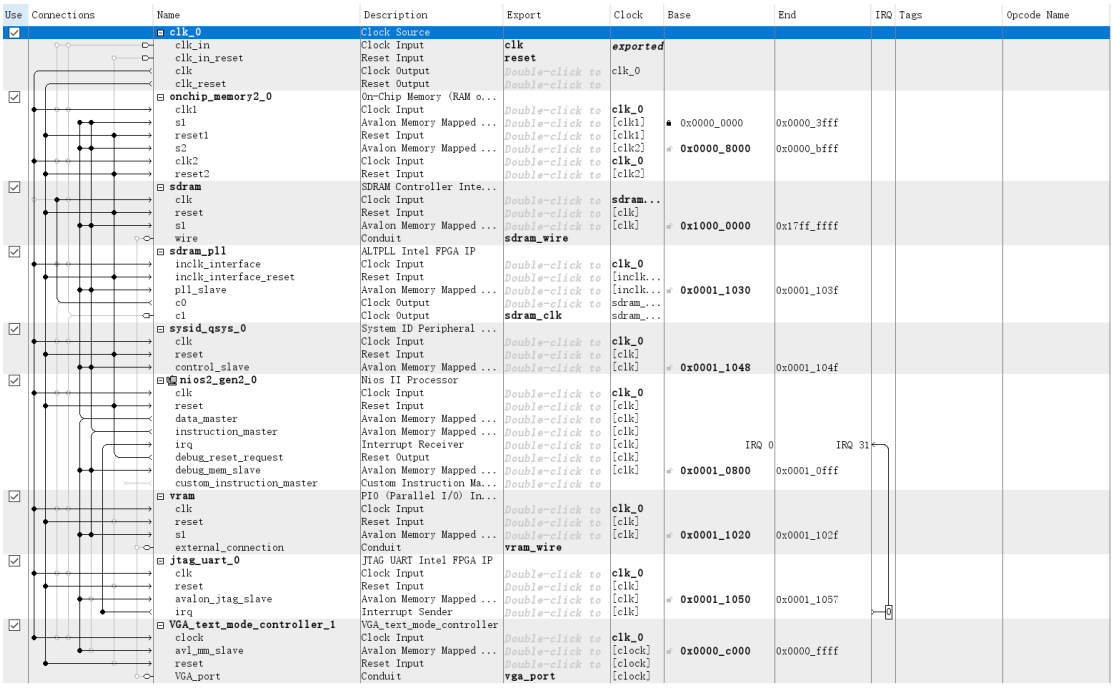

Fig-6: Our Connection in lab9_soc for week 2, notice we add one more clk signal to control the vram buffer.

5. Design Resources and Statistics

Resource	Utilization
LUT (Logic Elements)	30194
DSP (Embedded Multiplier 9-bit elements)	0
Memory (BRAM bits)	55296
Flip-Flop	21378
Frequency	36.69MHz
Static Power	106.35mW
Dynamic Power	233.05mW
Total Power	414.10mW

Table 1: Week 1.

Resource	Utilization
LUT (Logic Elements)	2554
DSP (Embedded Multiplier 9-bit elements)	0
Memory (BRAM bits)	340992
Flip-Flop	2483
Frequency	57.09MHz
Static Power	102.29mW
Dynamic Power	48.72mW
Total Power	226.37mW

Table 2: Week 2.

The resource utilization comparison between Week 1 and Week 2 demonstrates a big reduction in the use of Logic Elements, Flip-Flops, and an increase in Memory.

Week 1's design utilized 601 32-bit registers which increased compilation time due to the higher number of Logic Elements involved. In contrast, Week 2 implemented dual-port memory with 12-bit addressability, leading to fewer LUTs and Flip-Flops, enhancing the design's efficiency by increasing the operating frequency from 36.69 MHz to 57.09 MHz and reducing total power consumption from 414.10 mW to 226.37 mW.

However, Week 2's design, while more streamlined and efficient in resource usage, limits data manipulation to whole-register updates, restricting flexibility in certain use cases. Despite this, the advantages in terms of faster compilation and debugging efficiency make Week 2's design preferable for applications requiring higher speed and lower power usage, illustrating the impact of architectural optimizations on digital system performance.

6. Conclusion

We implemented the VGA Text Mode Controller with Avalon-MM Interface for both monochrome and color text displays. The primary challenge was integrating the on-chip memory for VRAM, which initially presented difficulties in memory management and addressing. The wrong way to calculate the address of each bit and the color map makes the screen messy. Also in week 1, the slow speed to compile brings us a huge challenge. Once we forgot to change the length of the avl_address.

A significant extension of this design could involve enhancing the graphical capabilities to include more complex sprite animations and possibly integrating user input to create interactive applications. The experience gained in managing VRAM and implementing the Avalon-MM interface will be invaluable for our final project, particularly in handling resource constraints and optimizing system performance.

This lab is quite significant for us to design a visual game in the final project. We learnt how to combine the memory, color mapper, vga_driver and vga_display for display something, and had a deeper understanding of the hardware-software interface.

8. References

[1] KTTECH. (2017, January 31). ECE 385 Lab 8: SoC with USB and VGA Interface in SystemVerilog. Retrieved from https://kttechnology.wordpress.com/2017/03/10/ece-385lab-8-soc-with-usb-and-vga-interface-in-systemverilog/. Teaching Assistant Blog
[2] ECE385 Faculty. (n.d.). Lab 9 description
[3] ECE385 Faculty. (n.d.). Introduction to SystemVerilog (pdf)
[4] ECE385 Faculty. (n.d.). Introduction to Quartus Prime in the lab manual.

9. Appendix A

lab9.sv

Module: lab9.sv
Inputs:

- **CLOCK_50**: System clock signal at 50 MHz.
- **[3:0] KEY**: 4-bit input for reset and other controls. Bit 0 is used as the Reset.
- **[12:0] DRAM_ADDR**: 13-bit output for SDRAM address.
- **[31:0] DRAM_DQ**: 32-bit bidirectional data bus for SDRAM.
- **[1:0] DRAM_BA**: 2-bit output for SDRAM bank address.
- **[3:0] DRAM_DQM**: 4-bit output for SDRAM data mask.
- **DRAM_RAS_N**: Output for SDRAM Row Address Strobe.

- **DRAM_CAS_N**: Output for SDRAM Column Address Strobe.
- **DRAM_CKE**: Output for SDRAM Clock Enable.
- **DRAM_WE_N**: Output for SDRAM Write Enable.
- **DRAM_CS_N**: Output for SDRAM Chip Select.
- **DRAM_CLK**: Output for SDRAM Clock.
- **[7:0] VGA_R**: 8-bit output for VGA Red.
- **[7:0] VGA_G**: 8-bit output for VGA Green.
- **[7:0] VGA_B**: 8-bit output for VGA Blue.
- **VGA_CLK**: Output for VGA Clock.
- **VGA_SYNC_N**: Output for VGA Sync signal.
- **VGA_BLANK_N**: Output for VGA Blank signal.
- **VGA_VS**: Output for VGA Vertical Sync signal.
- **VGA_HS**: Output for VGA Horizontal Sync signal.
- **[6:0] HEX0, HEX1**: 7-bit outputs for Hexadecimal display.

Outputs: None (All outputs are defined within input parameters)
Description: The **lab9** module is the top-level module for Lab 9, which integrates a VGA controller and SDRAM interface for use with a Nios II soft processor on the DE2-115 FPGA platform. It orchestrates the interface between FPGA peripherals, including SDRAM and VGA, facilitating the display of graphics and control of memory. This module initializes and manages the hardware settings and signal routing required to operate the FPGA's VGA display and memory interface.
Purpose: This module serves as the integration point for various subsystems in the Lab 9 project. It uses the system clock to synchronize the operations of memory access and video display. The module's implementation includes initializing the VGA display outputs and configuring the SDRAM for data storage and retrieval, which are crucial for the execution of Nios II processor tasks. The reset functionality is managed via the **KEY[0]** input, ensuring that the system can be reset asynchronously to address any operational anomalies.

hpi_io_intf.sv

Module: hpi_io_intf
Inputs:

- **Clk**: Clock input.
- **Reset**: Asynchronous reset input.
- **[1:0] from_sw_address**: Address lines from the switch interface.
- **[15:0] from_sw_data_out**: Data output from the switch to the EZ-OTG chip.
- **from_sw_r**: Read signal from the switch, active low.
- **from_sw_w**: Write signal from the switch, active low.
- **from_sw_cs**: Chip select signal from the switch, active low.
- **from_sw_reset**: Reset signal from the switch to the OTG chip, active low.

Outputs:

- **[15:0] from_sw_data_in**: Data input to the switch from the EZ-OTG chip.
- **[1:0] OTG_ADDR**: Address lines to the EZ-OTG chip.
- **OTG_RD_N**: Read signal to the EZ-OTG chip, active low.
- **OTG_WR_N**: Write signal to the EZ-OTG chip, active low.

- **OTG_CS_N**: Chip select signal to the EZ-OTG chip, active low.
- **OTG_RST_N**: Reset signal to the EZ-OTG chip, active low.
- **[15:0] OTG_DATA**: Bidirectional data bus between the NIOS II and the EZ-OTG chip.

Description: The **hpi_io_intf** module acts as an interface between the NIOS II processor and the EZ-OTG USB chip, managing communication via control signals and data transfer. It handles address decoding, data buffering, and control signal synchronization to facilitate data exchange between the system software and the external USB interface. The module controls data flow direction on the bidirectional **OTG_DATA** bus, ensuring that data integrity is maintained while preventing bus contention.
Purpose: The module operates under a simple protocol where control signals such as read/write, chip select, and reset are managed based on the system clock. Data from the switch is buffered in **from_sw_data_out_buffer** to drive the **OTG_DATA** bus when writing. When reading, the bus is set to high impedance state unless driven by external components. The module ensures proper initialization and signal integrity during data transfers by actively managing the tri-state bus and controlling signal edges in response to system reset or operational commands.

dualport.v

Module: dualport
Inputs:

- **[11:0] address_a, address_b**: Address inputs for port A and port B.
- **[3:0] byteena_a**: Byte enable for port A, controlling which bytes in the data words are active during write operations.
- **clock**: System clock signal.
- **[31:0] data_a, data_b**: Data inputs for port A and port B.
- **rden_a, rden_b**: Read enable signals for port A and port B, respectively.
- **wren_a, wren_b**: Write enable signals for port A and port B, respectively.

Outputs:

- **[31:0] q_a, q_b**: Data outputs for port A and port B.

Description: The **dualport** module implements a bidirectional dual-port memory interface using the **altsyncram** component, configured specifically for a Cyclone IV E FPGA. This module facilitates independent and simultaneous read and write operations on two separate ports (A and B), each capable of addressing up to 4096 words of 32-bit data. The operation mode is set to bidirectional dual port, allowing for versatile data handling and access patterns.
Purpose: The module supports byte-level write control through **byteena_a** for port A, and provides flexibility in data management with parameters allowing for bypass of clock signals on data inputs and outputs, aiming for minimal latency and optimal data throughput. The memory does not initialize on power-up, ensuring predictable behavior in system designs that require explicit control over memory states. The configuration settings for the module include specific read and write modes that handle new data with non-blocking read enabled, allowing efficient data processing in high-speed environments.

vga_controller.sv

Module: vga_controller
Inputs:

- **Clk**: 50 MHz system clock.
- **Reset**: Asynchronous reset signal.

Outputs:

- **hs**: Horizontal sync pulse, active low.
- **vs**: Vertical sync pulse, active low.
- **pixel_clk**: 25 MHz pixel clock output.
- **blank**: Blanking interval indicator, active low.
- **sync**: Composite Sync signal, active low (not used in this lab).
- **[9:0] DrawX**: Horizontal coordinate.
- **[9:0] DrawY**: Vertical coordinate.

Description: The **vga_controller** module is designed to generate timing and control signals necessary for interfacing with a standard 640x480 VGA display, operating slightly under the typical VGA frequency due to a 25 MHz pixel clock instead of the standard 25.175 MHz. This adjustment results in a slightly reduced refresh rate. The module manages horizontal and vertical sync pulses, as well as a blanking signal to denote visible and non-visible intervals. It also produces pixel clock and coordinate outputs that drive the display mechanism.
Purpose: This module counts through pixel coordinates and lines, resetting after reaching the end of each horizontal and vertical cycle. It generates a 25 MHz pixel clock by dividing the input 50 MHz clock. Horizontal and vertical sync pulses are managed according to VGA standards, but adjusted for the 25 MHz operation. The blanking signal is controlled to mask out non-visible portions of the display, improving the quality of the output on the VGA screen. The **sync** output is kept inactive as it is not required for this specific implementation but is included for completeness and potential future use.

vga_text_avl_interface.sv

Module: vga_text_avl_interface
Inputs:

- **CLK**: Clock input, typically 50 MHz used for VGA synchronization.
- **RESET**: Asynchronous reset signal.
- **AVL_READ**, **AVL_WRITE**: Avalon-MM read and write control signals.
- **AVL_CS**: Avalon-MM chip select.
- **[3:0] AVL_BYTE_EN**: Byte enable for Avalon-MM operations.
- **[11:0] AVL_ADDR**: Address for Avalon-MM access.
- **[31:0] AVL_WRITEDATA**: Data to write to memory or control register via Avalon-MM.

Outputs:

- **[31:0] AVL_READDATA**: Data read from memory or control register via Avalon-MM.
- **[3:0] red, green, blue**: Outputs to VGA color channels.
- **hs, vs**: Horizontal and vertical sync pulses for VGA.

- **sync, blank, pixel_clk**: Additional VGA control signals, including pixel clock and blanking signal.

Description:

The **vga_text_avl_interface** module interfaces with the Avalon Memory-Mapped (MM) bus to control a text mode VGA display on a DE2-115 board. It manages the VGA display's VRAM and a control register for handling screen settings. VRAM is organized in a raster order and supports 80x30 text mode. The module facilitates reading from and writing to VRAM and a control register based on Avalon-MM transactions. It decodes address inputs to direct operations either to VRAM or control registers and manages the VGA signal generation to display the contents stored in VRAM.

Purpose:

- The module processes read and write operations through the Avalon-MM interface, updating VRAM or the control register accordingly.
- It generates VGA signals based on the contents of VRAM, interpreting character data and control settings to render text on a VGA monitor.
- It supports direct hardware control over display attributes such as background and foreground colors, which can be dynamically adjusted via the control register.

font_rom.sv

Module: font_rom

Inputs:

- **[10:0] addr**: 11-bit address input to access font data.

Outputs:

- **[7:0] data**: 8-bit data output representing a row of pixels for a character glyph.

Description:

The **font_rom** module serves as a read-only memory (ROM) storage for font glyphs, providing pixel data for character rendering on a display. It is designed to hold a predefined set of graphical representations of characters, typically derived from a portion of the IBM codepage 437 set. Each character is represented in an 8x16 pixel format, with each row of the character represented by an 8-bit value, where each bit corresponds to a pixel (on or off).

Purpose:

- The module uses an 11-bit address input to access 8-bit data rows corresponding to specific parts of the glyphs stored in ROM.
- The ROM is defined as a parameterized array, containing predefined binary values that depict the visual representation of characters in a monochrome format. Each entry in the ROM corresponds to one row of a character glyph, allowing detailed and varied character designs.
- The data output provides the pixel row data for the addressed character row, which can be used directly by display controllers or other graphic processing units.

This module is essential for systems requiring fixed-font text rendering, such as information displays, user interfaces, or any application needing basic text output capabilities without the overhead of dynamic font loading. The **font_rom** ensures fast access to glyph data, facilitating efficient text rendering by directly mapping character codes to their graphical representations.