**California State University, Fresno**

**Lyles College of Engineering**

**Electrical and Computer Engineering Department**

**TECHNICAL REPORT**

**Experiment Title:** Software Design with the NIOS II Processor, Final Project

**Course Title:** ECE 178 Embedded Systems

**Date Submitted:** May 6, 2015

**Honor Code Statement:**

**“I have done my own work and have neither given nor received**

**unauthorized assistance on this work.”**

|  |  |
| --- | --- |
| **Prepared By:** |  |
| **Christopher Hays** |  |
|  |  |
| **Signature:** |  |

**INSTRUCTOR SECTION**

**Comments:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**TABLE OF CONTENTS**

**Section:** **Page:**

Title Page ……………………………………………………………………...…. 1.

Table of Contents ……………………………………………………………….... 2.

1. Statement of Objectives ……………………………………………………….. 3.

2. Theoretical Background …………………….…………………………………. 3.

3. Experimental Procedure ………………………………………………………. 3.

3.1 Equipment Used ……………………………………………………... 3.

3.2 Laboratory Procedure ……………………………………………...… 4.

4. Analysis ……………………………………………………………………...... 6.

5. Conclusions ……………………………………………………………………. 9.

6. Appendix ………………………………………………………………………. 10.

**1. STATEMENT OF OBJECTIVES**

The objective of this assignment was to design an embedded system using the NIOS II processor, which would read a file from an SD card (using SPI protocol) and output the contents of the file to a VGA display. This embedded system was implemented on the DE2 development board, using the custom system that has been designed over the course of the semester and adding components specific to the SD card and the VGA control. The program uses polling to verify the existence of an SD card then reads a specific sector of the card that contains the data file. The data buffer is then output to the VGA monitor character buffer for display.

**2. THEORETICAL BACKGROUND**

The NIOS II is a 32-bit embedded processor designed for the Altera family of FPGAs. The *Altera Monitor Program* is used to specify the NIOS II build, compile, and load any user-defined programs. This program visually represents the register and memory contents, and provides debugging tools. *Qsys* is a system integration tool that represents the various components of a system graphically, creating an easy way to make/visualize connections between components. Every piece of the system is modeled and libraries of pre-made IP cores are available to be added to a custom system. Once the system components are connected to each other, *Qsys* is used to generate Verilog code and block diagrams to be used within *Quartus II.*

Secure Digital (SD) cards are a non-volatile form of memory, typically used in portable devices. SD cards are typically interfaced with in one of three ways:

* SPI Bus Mode
* One-bit SD Bus Mode
* Four-bit SD Bus Mode

This lab made use of SPI Bus Mode, which is a synchronous serial communication interface typically used in embedded systems. It uses a master/slave setup and sends a frame of information at a time using full duplex.

**3. EXPERIMENTAL PROCEDURE**

**3.1 Equipment Used**

Altera Monitor Program Software

Altera DE2 FPGA Development Board

Quartus II Software

Qsys

NIOS II CPU

USB Blaster

HxD Hex Reader Software

**3.2 Laboratory Procedure**

The first step was to setup the appropriate hardware IP cores using QSys. The SD card interface uses a 512-byte buffer to read and write from the card. The CMDARG register is used to store the address of the desired sector while the CMD register can take the read or write command. When the write command is sent, the 512-byte buffer is written to the current sector of the card and, conversely, when the read command is sent, the buffer fills with the contents of the sector.

Setting up the VGA required the use of several IP cores. A PLL provided by the Altera University Program was used to supply a 25 MHz clock for the VGA (Figure 1). An AV CONFIG core was used to setup the VGA, this ran when the system started and required a delay in software to make sure that all the components were ready before display. The main VGA controller makes use of a Dual-Clock FIFO, a Character Buffer, Pixel Scalar, RGB Resampler, and Pixel Buffer. Figures 2 through 5 show the connections in QSys. The end result is a memory-mapped buffer that will write characters to the VGA display, with the address corresponding to the characters position on the screen. The VGA Controller and FIFO are connected to the 25Mhz vga\_clk.

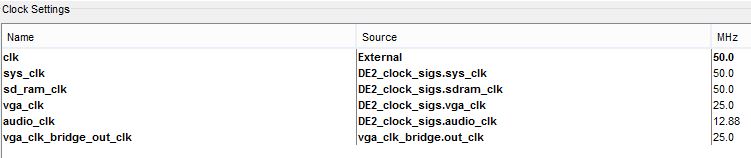


Figure 1: PLL Clocks

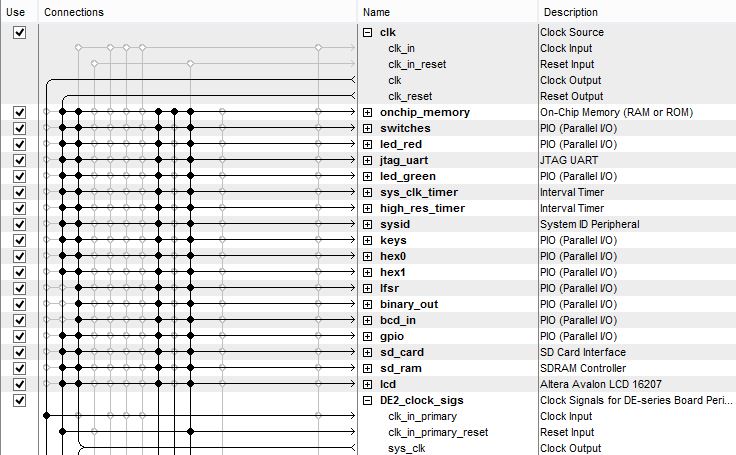


Figure 2: Qsys System

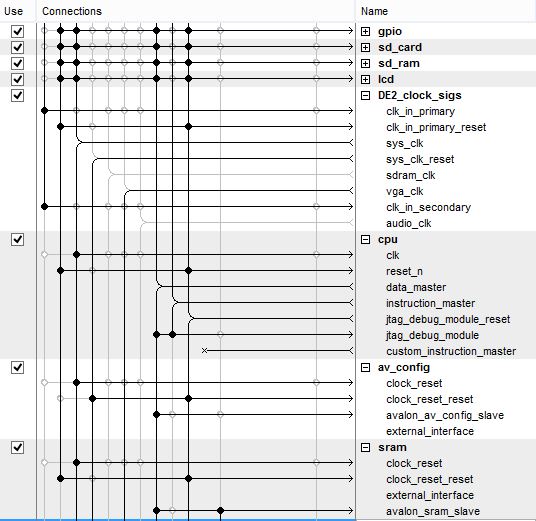


Figure 3: QSys System

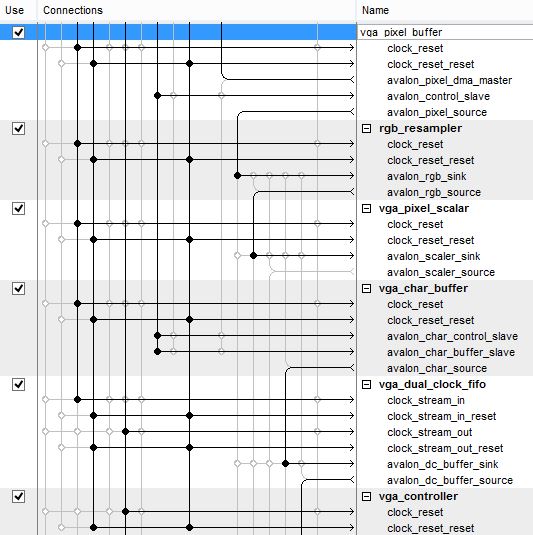


Figure 4: QSys System

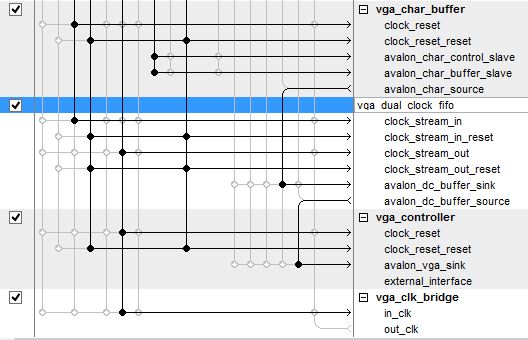


Figure 5: QSys System

The main section of the program performs three functions: read the SD card, call a delay, and display the contents on the VGA monitor. The delay is needed to make sure the video components have been set up. A function was implemented to write to the SD card, but is not used; a text file was written to the SD card in Windows to serve as a data file. The HxD Hex Reader program was used to determine that this file is stored at sector 768 on the card.

The Read SD function begins by reading the ASR of the SD controller IP core and outputting the value to the red LEDs for debugging purposes. The function loops, reading the ASR until bit 1 is set, meaning there is an SD card present. The desired sector (768 in this case) is shifted logical left by 9, essentially multiplying by 512: this is the address of the sector on the SD card which is written to the CMDARG register. Next, the read command (0x11) is sent to the CMD register, ordering the controller to read the sector, filling the 512-byte buffer. The subroutine now loops, checking bit 2 of the ASR, waiting for the current operation to complete.

The Display function begins by clearing the index that will be used to keep track of the current position within the SD card data buffer. Next, a loop is entered that will write characters until it has reached the end of the 512-byte buffer. First, the output cursor position is calculated. The cursor works on an (x,y) coordinate system, with (1,1) being the top left corner of the screen. Registers are used to hold the x value and the y value, and these are used to calculate the next desired cursor position using the formula: cursor = (x + 128\*y).

The first character in the buffer is output to this position while the x-position and buffer index are incremented. The y-position is incremented every 80 characters, due to the width of the VGA display. When 512 characters have been written the display function is done.

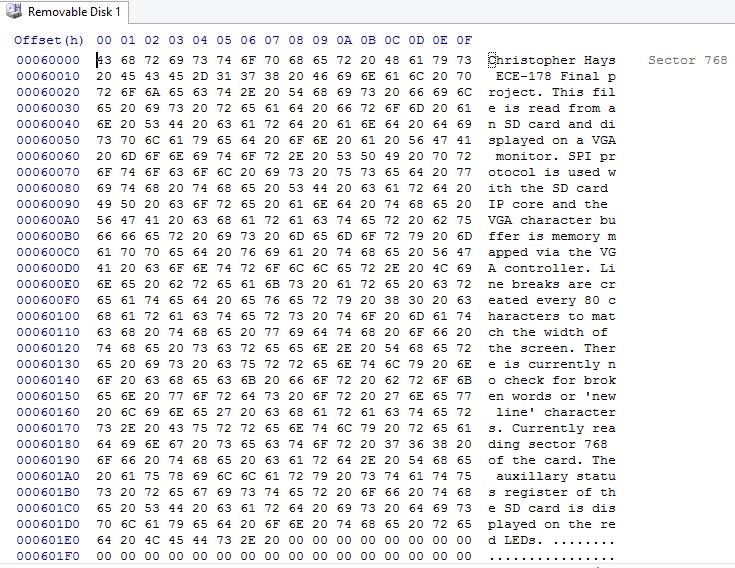


Figure 6: SD Card Contents at Sector 768

**4. ANALYSIS**

The Quartus II and QSys project files compiled and downloaded to the development board without any issues, as did the assembly code (Figures 7-10). To test the program, the SD card was removed before running it. The ASR of the SD control module was displayed on the red LEDs while the VGA output remained blank. The SD card was then inserted and the file contents at sector 768 were immediately displayed on the monitor.

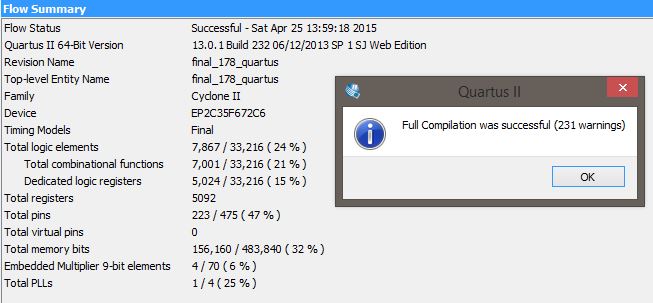


Figure 7: Successful Quartus II Compilation

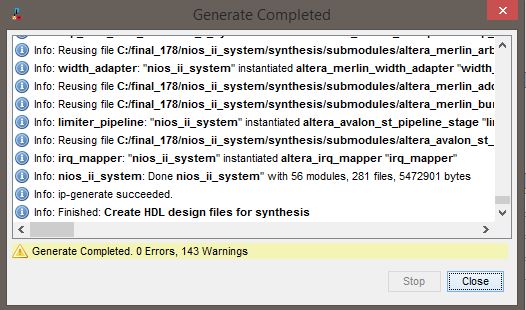


Figure 8: Successful QSys Generation

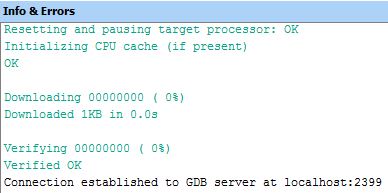


Figure 9: Assembly File Compilation

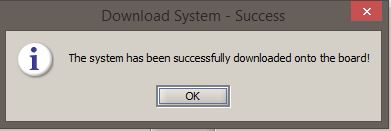


Figure 10: System Download

Figure 11 shows a section of the program’s final output. The SD controller buffer contents have been printed on the screen; if the file were larger, here the sector number would be incremented before calling another read subroutine. The x and y coordinates would pick up from their current position, creating a seamless display of text. Fully commented assembly code is shown in the Appendix.

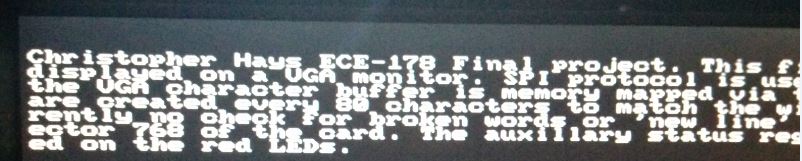


Figure 11: Successful VGA Output

**5. CONCLUSION**

This assignment demonstrated the ability to read from an SD card and write the contents to a VGA monitor. The SD controller IP core creates a memory-mapped 512-byte buffer that can be read from or written to, essentially eliminating the need to know the details of SPI protocol. Though, these details are important for a full understanding of the hardware. The VGA controller required a lot of setup time and was modeled after the Altera University Program’s DE2 Media Computer, though the Alpha Blending core was removed because it created errors. After the hardware setup and clock synchronization via the DE2 Clock Signals PLL, the VGA buffer is also memory-mapped and writing to it becomes trivial. This lab also had the benefit of teaching buffer control and operations to avoid overflow or having a pointer that is out of bounds.

**6. APPENDIX**

**The Assembly Code:**

/\* Christopher Hays \*/

/\* ECE 178 Final Project \*/

/\* Spring 2015 \*/

/\* SPI interface and SD card \*/

/\* Read a sector of the card and output to vga \*/

/\*\*\*\*\*\*\*\* RESET VECTOR \*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.section .reset, "ax" /\* label the reset vector \*/

reset:

movia sp, 0xff0 /\* the end of the stack \*/

br \_start /\* branch to start \*/

/\*\*\*\*\*\*\*\* EXCEPTION VECTOR \*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.section .exceptions, "ax" /\* label the exception vector \*/

exception\_handler:

addi sp, sp, -0x4 /\* allocate stack \*/

stw ra, 0(sp) /\* store ra \*/

exceptions\_done:

ldw ra, 0(sp) /\* restore the return address \*/

addi sp, sp, 0x4 /\* de-allocate stack \*/

eret /\* return from the exception handler \*/

/\*\*\*\*\*\*\*\* CONSTANTS \*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.equ switches, 0x24c0

.equ led\_red, 0x24d0

.equ jtag, 0x24e8

.equ led\_green, 0x24b0

.equ hex0, 0x24a0

.equ hex1, 0x2490

.equ keys, 0x2480

.equ sram, 0x100000

.equ timer, 0x2420

.equ timer\_2, 0x2400

.equ lfsr, 0x2470

.equ binary\_out, 0x2460

.equ bcd\_in, 0x2450

.equ gpio, 0x2440

.equ sd\_card, 0x2000

.equ sd\_ram, 0x800000

.equ lcd, 0x1000

.equ char\_addr, 0x4000

/\*\*\*\*\*\*\*\* START \*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.global \_start

\_start:

movia r2, 0x1 /\* a register to hold a 1 constant \*/

movia sp, 0xff0 /\* create a stack pointer address \*/

movia r3, 0x0 /\* register for computation results \*/

movia r4, 0x0 /\* return subroutine values here \*/

movia r6, sd\_card /\* the sd card base address \*/

movia r7, led\_red /\* red led base address \*/

movia r20, 768 /\* the current sector of the sd card \*/

movia r21, 0x0 /\* sd buffer byte index \*/

movia r9, char\_addr /\* base address of the vga character buffer \*/

movia r15, 0x1 /\* cursor x position \*/

movia r16, 0x1 /\* cursor y position \*/

slli r10, r16, 7 /\* multiply y by 128 \*/

add r10, r10, r15 /\* add x to this, starting at position 1,1 (x,y) \*/

add r9, r9, r10 /\* add to base address \*/

/\* position formula is (x + 128\*y) \*/

/\*\*\*\*\*\*\*\* MAIN \*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

main:

call read\_sd /\* read the sd card into the buffer \*/

call delay /\* delay \*/

call display /\* output to the vga \*/

end\_main:

br end\_main

/\*\*\*\*\*\*\*\* SUBROUTINES \*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.global delay

delay:

addi sp, sp, -0x4 /\* allocate stack \*/

stw ra, 0(sp) /\* store ra \*/

movia r3, 1250000

delay\_loop: /\* delay loop \*/

subi r3, r3, 1

bne r3, r0, delay\_loop

ldw ra, 0(sp) /\* restore the return address \*/

addi sp, sp, 0x4 /\* de-allocate stack \*/

ret

.global read\_sd /\* read the sd card \*/

read\_sd:

addi sp, sp, -0x4 /\* allocate stack \*/

stw ra, 0(sp) /\* store ra \*/

verify:

ldwio r3, 564(r6) /\* read the ASR of the SD Card IP Core \*/

stwio r3, 0(r7) /\* output to red leds for debugging \*/

andi r3, r3, 0x2 /\* check for bit 1 \*/

beq r3, r0, verify /\* loop if no card present \*/

mov r3, r20 /\* we want to read the current sector \*/

slli r3, r3, 9 /\* multiply by 512: shift logical left \*/

stwio r3, 556(r6) /\* write this address to the CMDARG register \*/

movia r3, 0x0011 /\* read\_block command \*/

stwio r3, 560(r6) /\* write to CMD reg \*/

card\_busy:

ldwio r3, 564(r6) /\* read the ASR \*/

stwio r3, 0(r7) /\* output to red leds for debugging \*/

andi r3, r3, 0x4 /\* check for bit 2 \*/

bne r3, r0, card\_busy /\* loop if busy \*/

ldw ra, 0(sp) /\* restore the return address \*/

addi sp, sp, 0x4 /\* de-allocate stack \*/

ret

.global write\_sd /\* write to the sd card \*/

write\_sd:

addi sp, sp, -0x4 /\* allocate stack \*/

stw ra, 0(sp) /\* store ra \*/

verify\_write:

ldwio r3, 564(r6) /\* read the ASR of the SD Card IP Core \*/

stwio r3, 0(r7) /\* output to red leds for debugging \*/

andi r3, r3, 0x2 /\* check for bit 1 \*/

beq r3, r0, verify\_write /\* loop if no card present \*/

mov r3, r20 /\* we want to write to the current sector \*/

slli r3, r3, 9 /\* multiply by 512 \*/

stwio r3, 556(r6) /\* write this address to the CMDARG register \*/

movia r3, 0x0018 /\* write\_block command \*/

stwio r3, 560(r6) /\* write to CMD reg \*/

card\_busy\_write:

ldwio r3, 564(r6) /\* read the ASR \*/

stwio r3, 0(r7) /\* output to red leds for debugging \*/

andi r3, r3, 0x4 /\* check for bit 2 \*/

bne r3, r0, card\_busy\_write /\* loop if busy \*/

ldw ra, 0(sp) /\* restore the return address \*/

addi sp, sp, 0x4 /\* de-allocate stack \*/

ret

.global display /\* writes the buffer to the vga \*/

display:

addi sp, sp, -0x4 /\* allocate stack \*/

stw ra, 0(sp) /\* store ra \*/

movia r21, 0x0 /\* buffer byte index \*/

write\_char:

add r4, r21, r6 /\* add buffer index to base address of buffer \*/

ldbio r3, (r4) /\* read where the buffer index is pointing \*/

/\* calculate cursor position \*/

slli r10, r16, 7 /\* multiply y by 128 \*/

add r10, r10, r15 /\* add x to this \*/

add r10, r9, r10 /\* add to base address \*/

stbio r3, 0(r10) /\* output character to vga \*/

addi r15, r15, 1 /\* increment the cursor x position \*/

subi r3, r15, 79 /\* check if 79 characters have been \*/

bne r3, r0, continue\_write /\* written to this line \*/

addi r16, r16, 1 /\* if so, increment y coordinate \*/

movia r15, 0x1 /\* set x coordinate to 1 \*/

continue\_write:

addi r21, r21, 1 /\* increment the buffer index \*/

subi r3, r21, 512 /\* buffer index - 512 \*/

bne r3, r0, write\_char /\* if zero we are done with the sector\*/

ldw ra, 0(sp) /\* restore the return address \*/

addi sp, sp, 0x4 /\* de-allocate stack \*/

ret

.end

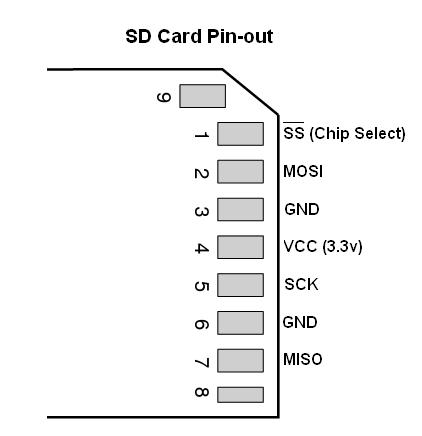


Figure 12: SD Card Pinout

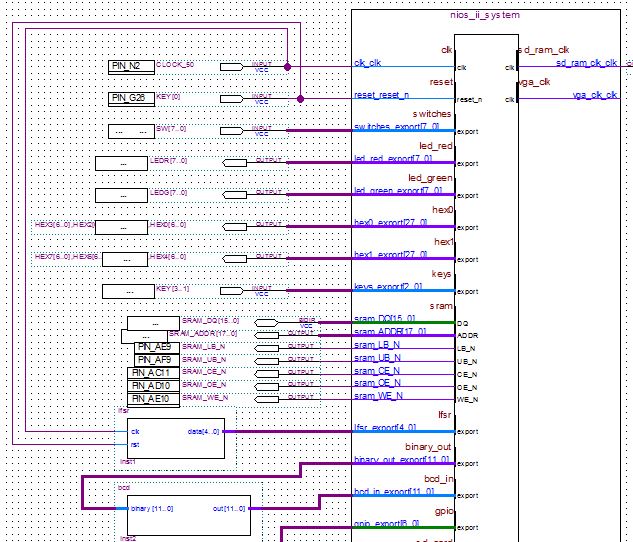


Figure 13: Quartus System Part 1

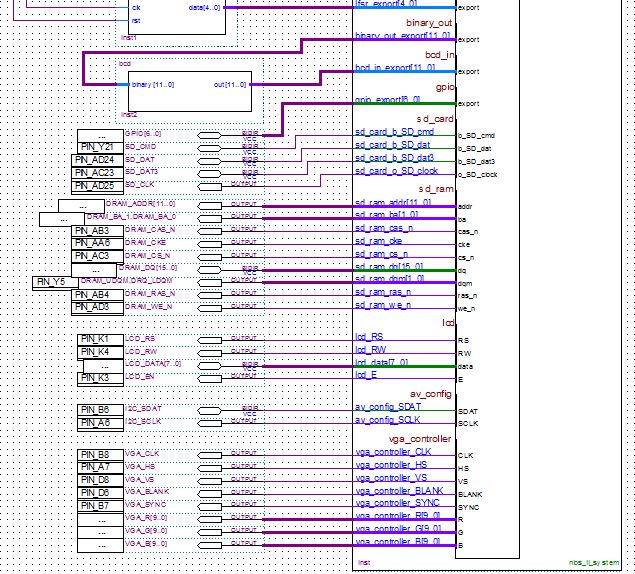


Figure 14: Quartus System Part 2