EE/ CSE 371 Lab Report 4 & 5

Ahmed Moalim

Radleigh Ang

Arvin Tang

Abstract

In lab 4, we developed a NIOS II microprocessor using Qsys in Quartus. We then wrote simple C programs in Eclipse to run on our microprocessor. This allowed us to practice communicating between the processor and the hardware, by generating commands from the microprocessor to our modules and sending signals from the modules back to the microprocessor.

In lab 5, we continued working with the QSys tools and concepts from lab 4 and created an asynchronous serial network to transfer data from our FPGA to the console and back. We then worked with another group to transmit data between the two boards, displaying the ASCII character representation of data we received on our Eclipse console. As extra credit we then implemented a Simon game using the NIOS II microprocessor and a VGA driver to display the game on the monitor.

Table of Contents

1. Introduction

In Lab 4, we designed and tested a NIOS II microprocessor on the DE1 SoC board. The NIOS II microprocessor was designed within the Qsys environment of Quartus. First, we ran simple C template programs on the microprocessor (Count Binary, Lights and Switches, Hello World Small). Then we modified Hello World Small to accept user inputs in the Eclipse console, to activate the switching behaviour, and to read board information to modify behavior, using a switch to complement the normal behavior. Ultimately, this lab gave us a chance to communicating between the Eclipse console and the board's inputs and outputs. The lab illustrated how the verilog modules, the NIOS II microprocessor, and the C program all interact with one another.

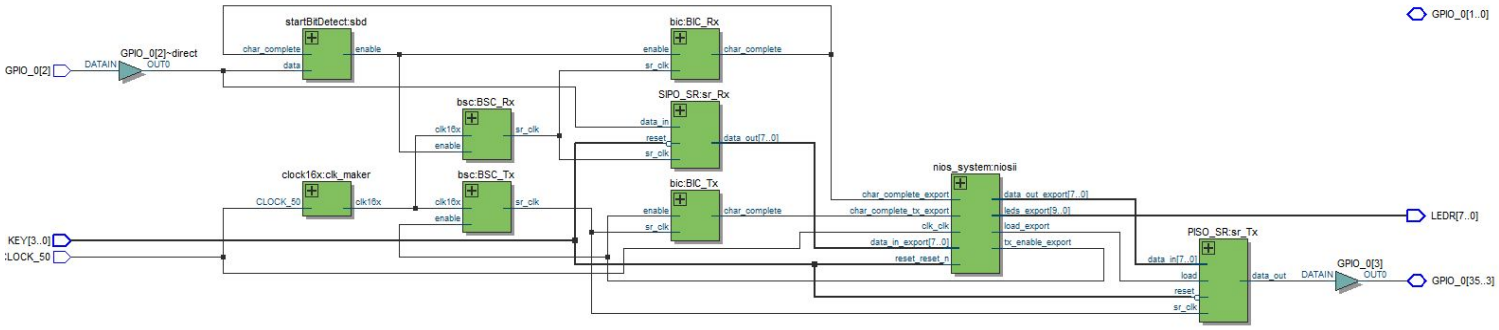
In Lab 5, we designed a serial communication system for our microprocessor so that we may communicate with another group's board. The data is turned into parallel data for storage on each board's but is transmitted serially. The data we received is then displayed on the Eclipse console for the user to see. To take the design further we then developed and designed a Simon game that takes user input from the boards keys and uses it to play a color matching game displayed on the VGA.

2. System Description

The following sections describe each of our hardware and software modules, as well as providing the inputs and outputs to/from our system.

2.1 Serial Communication

2.1.1 DE1_SoC



This DE1_SoC is the top level module that instantiates the submodules for the serial communication system.

2.1.2 clock16x

clock16x generates a clock signal that is 16 times faster than the Baud rate of our

communication system, which is 9600 bits per second. This generated clock is called clk16x.

- inputs: CLOCK_50 (50 MHz clock)
- outputs: clk16x

2.1.3 startBitDetect

startBitDetect reads the data line and looks for the start bit of a data frame. When the start bit has been detected, this module will signal the other modules in the receiver to begin processing the serial data until the data is fully processed.

- inputs: data (serial in data), char_complete
- outputs: enable

2.1.4 bsc

The bsc module uses the 16-times-Baud-rate clock signal (clk16x) to generate a clock signal with frequency equal to the Baud rate of our communication system. We call this signal sr_clk. sr_clk is generated such that its positive edges coincide with the middle of a bit on the serial data line.

The bsc module is instantiated twice in our system, one for the receiver and one for the transmitter. The difference between the two is where their enable signals comes from. For the receiver, the enable signal comes from startBitDetect. For the transmitter, the enable signal comes from software on the Nios II processor.

- inputs: clk16x, enable
- outputs: sr_clk

2.1.5 bic

The bic module takes the sr_clk and the enable signal and outputs a char_complete signal. The char_complete signal becomes true after a data frame has been fully processed.

The bsc module is instantiated twice in our system, one for the receiver and one for the transmitter. The difference between the two is where their enable signals comes from. For the receiver, the enable signal comes from startBitDetect. For the transmitter, the enable signal comes from software on the Nios II processor.

- inputs: sr_clk, enable
- outputs: char_complete

2.1.6 SIPO_SR

SIPO_SR stands for serial in, parallel out shift register. This module is used for receiving data serially and then passing it to the Nios II processor parallelly. Its inputs are the sr_clk generated by the bsc module, the serial data line, and a reset signal. The output is an 8-bit parallel data bus.

- inputs: sr_clk, reset, data_in (1 bit, serial)
- outputs: data_out (8 bits)

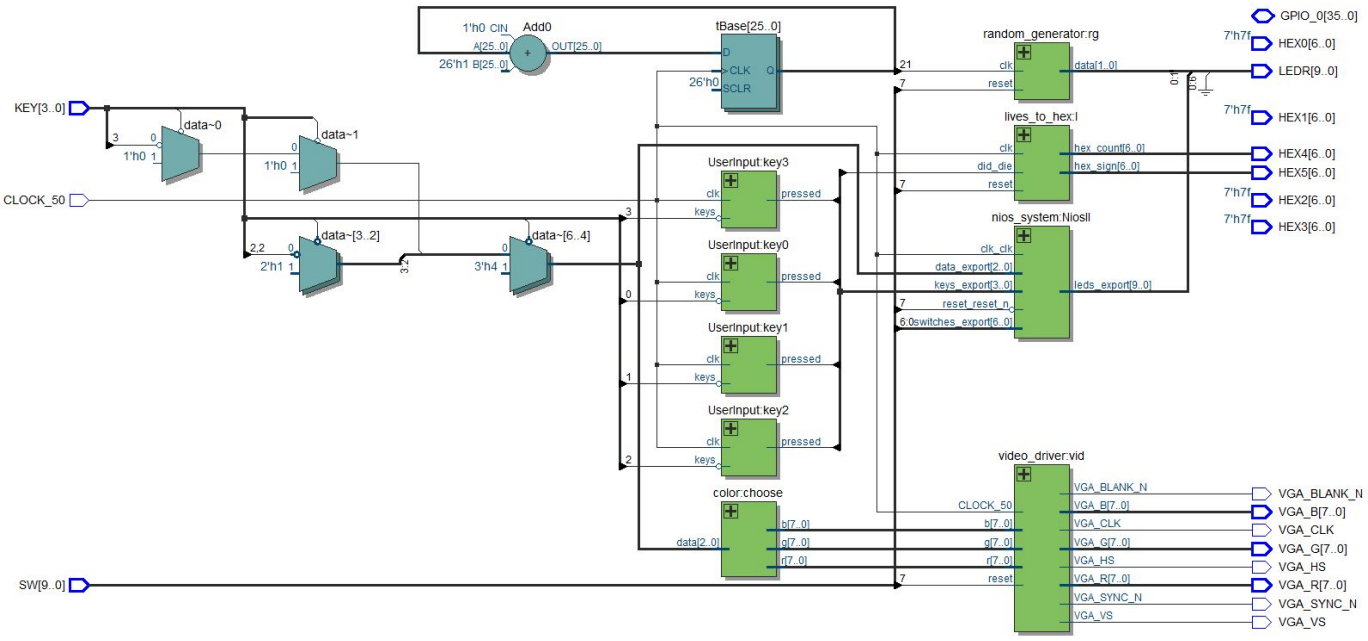
2.1.7 PISO_SR

PISO_SR stands for parallel in, serial out shift register. This module is used for transmitting data serially. The data it transmits comes from the Nios II processor in the form of an 8-bit parallel data bus. Its inputs are the sr_clk, reset, a load signal from the Nios II processor, and the 8-bit data bus also from the Nios II processor. The output is a single bit of the data passed to the serial output of the communication system.

- inputs: sr_clk, reset, load, data_in (8 bits)
- outputs: data_out (1 bit, serial)

2.2 Budget Simon (Extra Credit)

2.2.1 DE1_SoC_extra



This is DE1_SoC_extra is our top level module that instantiates the submodules for the Simon system.

2.2.2 User Input (Not Demoed)

User Input was implemented as an edge detector, designed to avoid any metastability from the KEYs as well as being high for only single pulse. The output, “pressed”, is high for only one clock cycle.

- inputs: clk, reset, keys

- outputs: pressed

The idea for the User Input was to ensure that the user's key press only counts once, which will allow them to match the colors.

2.2.3 Lives to Hex

lives_to_hex.sv was implemented as with a counter and driver which translates the count into a hex or life value. The counter is incremented only if the input did_die is high,

- inputs: clk, reset, did_die
- outputs: hex_count, hex_sign

This served as a "score" that keeps track of the user's total lives.

2.2.4 Random Number Generator

lives_to_hex.sv was implemented as a 4 bit linear-feedback shift register (LFSR), the output was then mod by 4.

- inputs: clk, reset
- outputs: data

The random number generator was eventually supposed to generate a random sequence of colors that the user will have to match with their key presses.

2.2.5 Color

Color is a simple case module, which takes in a 3 bit number and outputs the corresponding rgb value for the Video Driver module.

- inputs: data
- outputs: r, g, b

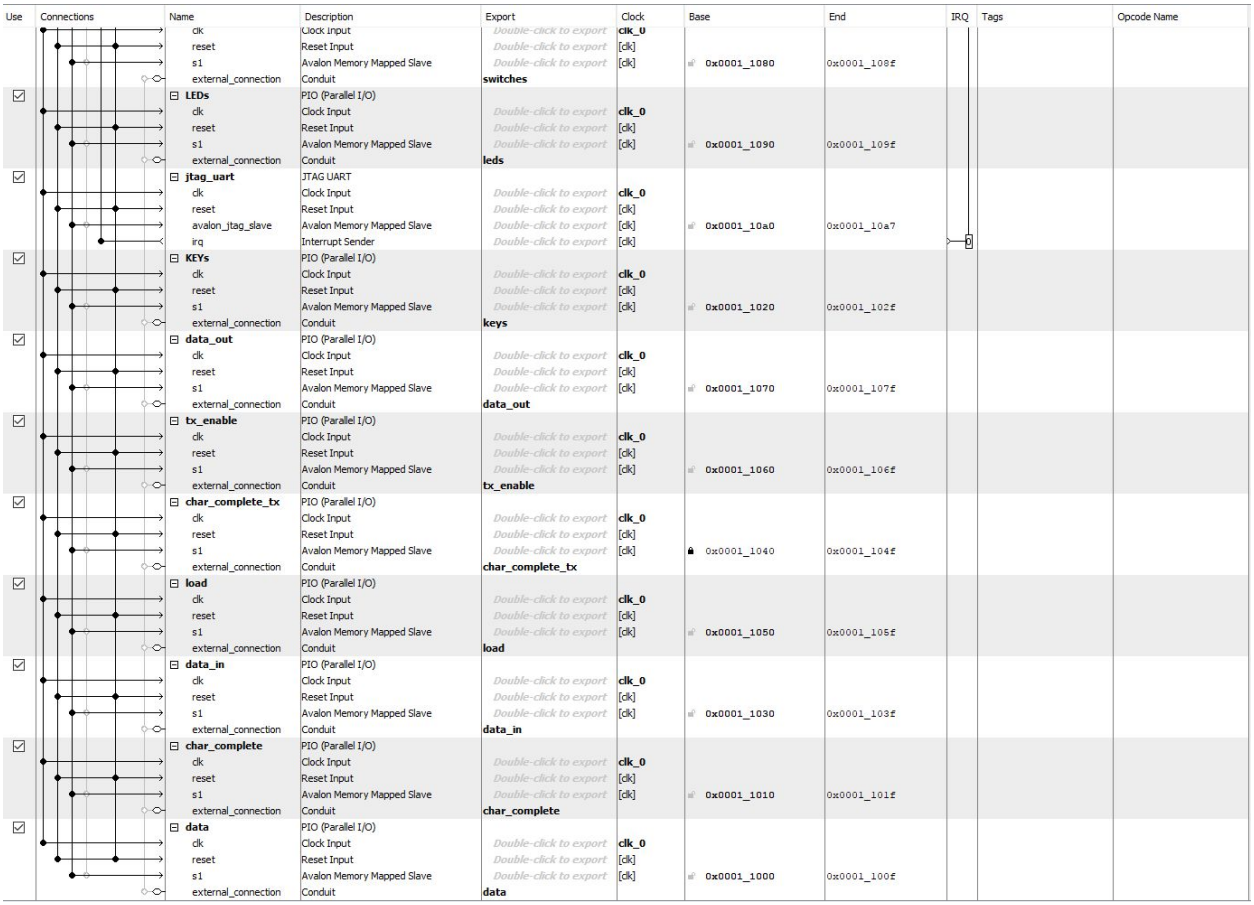
This module would serve the basis for both the computer's sequence, and the sequence flashed by the user.

2.2.6 Video Driver/altera_up_avalon_video_vga_timing

The Video Driver and altera_up_avalon_video_vga_timing were default modules created by Altera. We used these modules to control each pixel of color and their positions on the screen.

2.3 NIOS II Microprocessor and Qsys

The NIOS II Microprocessor was the interface between the FPGA and the C programming software in Eclipse.



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1080	0x0001_108f			
		external_connection	Conduit	switches						
☑		**LEDs**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1090	0x0001_109f			
		external_connection	Conduit	leds						
☑		**jtag_uart**	JTAG UART							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_10a0	0x0001_10a7			
		irq	Interrupt Sender	Double-click to export	[clk]			0		
☑		**KEYs**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1020	0x0001_102f			
		external_connection	Conduit	keys						
☑		**data_out**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1070	0x0001_107f			
		external_connection	Conduit	data_out						
☑		**tx_enable**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1060	0x0001_106f			
		external_connection	Conduit	tx_enable						
☑		**char_complete_tx**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1040	0x0001_104f			
		external_connection	Conduit	char_complete_tx						
☑		**load**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1050	0x0001_105f			
		external_connection	Conduit	load						
☑		**data_in**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1030	0x0001_103f			
		external_connection	Conduit	data_in						
☑		**char_complete**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1010	0x0001_101f			
		external_connection	Conduit	char_complete						
☑		**data**	PIO (Parallel I/O)							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1000	0x0001_100f			
		external_connection	Conduit	data						

The above image is a snapshot of the NIOS system map that we used for both the serial communication and the Simon Game.

The Nios mapped the following:

- switches (inputs)
- LEDs (outputs)
- KEYs (inputs)
- data_out (output)
- tx_enable (output)
- char_complete_tx (input)
- load (output)
- data_in (input)
- char_complete (input)
- data (output, used in Simon game)

3. Hardware Description

3.1 Serial Communication

3.1.1 DE1_SoC

DE1_SoC is the top level module of our communication system. It instantiates the submodules for the communication system.

3.1.2 clock16x

clock16x uses the FPGA board's 50 MHz clock to generate a clock signal that is 16 times faster than the Baud rate of our communication system, which was 9600 bits per second. The math with the frequencies shows that the 16-times-Baud-rate clock signal must change value every 162 positive clock edges on the 50 MHz clock. This clock is used to determine when to sample the serial data line, ideally, at the middle of a bit.

3.1.3 startBitDetect

startBitDetect takes the serial data line and a char_complete signal as input. The output is an enable signal that allows other modules to proceed with processing the serial data. startBitDetect will look for the start bit of data frame on the serial data line. When the start bit has been detected, enable becomes true and stays true until the char_complete signal indicates that the data frame has been fully processed.

3.1.4 bsc

The bsc module takes the 16-times-Baud-rate clock signal (clk16x) generated by the module clock16x and outputs a signal that we call sr_clk. sr_clk is the clock signal with frequency equal to the Baud rate. We generate sr_clk such that its positive edge occurs at the middle of a bit on the serial data line. Since clk16x oscillates 16 times during the time a single bit is present on the data line, we can make the sr_clk signal transition from 0 to 1 after 8 oscillations of clk16x to create the desired clock signal for receiving data serially.

3.1.5 bic

The bic module takes the sr_clk and the enable signal and outputs a char_complete signal. The char_complete signal becomes true after a data frame has been fully processed, which happens after 11 positive edges on the sr_clk. We keep track of this count with a counter in the module.

3.1.6 SIPO_SR

This module is a serial in, parallel out shift register. On each positive edge of the sr_clk, each bit in an internal register get shifted to the right by one position. The leftmost position will be filled by the value currently on the serial data line. This internal buffer is 10 bits wide to store 8 data bits, a start bit, and a stop bit. The output is the 8 bits in the middle of the register. This module has an asynchronous reset, which sets all of the bits in

the shift register to 1.

3.1.7 PISO_SR

This module is a parallel in, serial out shift register. When no data is being transmitted, the PISO_SR continually outputs a 1. When data is present and the load signal is high, the shift register will a load a 0 for the start bit, the 8 data bits, and a 1 for the stop bit. Then, on each positive edge of the sr_clk, the leftmost data bit gets shifted out of the register. This module has an asynchronous reset, which sets all of the bits in the shift register to 1.

3.2 Budget Simon (Extra Credit)

3.2.1 DE1_SoC Extra

This module was designed to be the top level firmware control of the Simon Game. It instantiates the colors, the user input, the lives counter, and the random number generator. The DE1_SoC also currently has an always block which maps the KEYS to a data number, which is passed to the Color module to display onto the VGA.

3.2.2 User Input

User Input was implemented as an edge detector and to avoid metastability. To avoid metastability, we assigned our raw key input with two different flip flops to delay it and ensure a clean signal. We then “and” that clean signal with the “not” of the delayed version, creating a pulse that we return back to the user. The delayed signal cancels out any remaining high values after the delay.

3.2.3 Lives to Hex

This module was implemented using a case statement to translate the internal 4 bit count into a 7-bit signal for the hex_count and hex_sign. Count ranges from 0 to 10 and each case is used to translate to the corresponding hex_count and hex_sign ranging from 2 to -8. Count as was mentioned earlier is a internal 4 bit number that is incremented by a D Flip Flop on the positive edge of the clock. The value only increments if the input did_die is high and once count reaches a value of 10 then it is reseted to 0, so that it may recount.

3.2.4 Random Number Generator

This module was implemented using the concept of an LSFR we designed a simple shift register with feedback from two bits, in the register chain. These taps in this module are at bit 1 and bit 4, and are references as [4,1]. The data input to the LFSR is generated by XOR-ing the tap bits; the remaining bits function as a standard shift register. The sequence of values generated by an LFSR was determined by the tap selections of bit 4 and 1 with it repeating every 8th time in the cycle. Once the output was shifted we then mod the value by 4 so that the LSFR value will not be greater than 3 and will match the 2’bit constraint of the output.

3.2.5 Color

This module was implemented using a case statement that took in a 3 bit number, which is currently taken from the always block in the DE1_SoC_extra module. The r, g, and b values are set from 0 to 255, depending on the 3 bit number input. In order to get the yellow display on the VGA, we set 255 to both red and green. Setting to red, green, and blue are self-explanatory.

4. Software Description

4.1 Lab 4

4.1.1 Count Binary

count binary.c is from the Eclipse template and was used to explore basic NIOS to C interaction.

4.1.2 Lights and Switches

lights and switches.c is from page 26 of the Introduction to Qsys Tool.pdf. It was used to explore the use of base addresses, and the need to declare variables as (volatile char *) to temporarily disable compiler optimization. As well as having the proper reset condition when loading the program onto the board. We then implemented a complement version that uses a single switch as an identifier that when this switch is high the lights become the compliment of all the switches.

4.1.3 Hello World Small

hello world small.c is from the Eclipse template and was used to explore displaying a message from the NIOS II to the console. The modified version of hello world small that accepts a user input, which in turn activates the behavior seen in the lights and switches program. Additionally, turning on SW0 complements that behaviour. It is worth noting that the alt form of the standard i/o had to be used, since the NIOS II we built had limited memory capacity.

4.2 Lab 5

4.2.1 Serial Communication - serial_com.c

The C file that interfaces with our serial communication system prompts the user for a character to send and then displays the character received in the receiver. Due to time constraints, we were not able to finish debugging the hardware's receiver modules. Consequently, serial_com.c prints out the value of the character received in hexadecimal for debugging purposes. An output sequence of serial_com.c might look like:

```
Type a character to send: C
Character has been transmitted
Character received:
c2
```

```
Type a character to send: s
Character has been transmitted
Character received:
ffffffff
```

where the underlined text is user input. This example came from a run of our program during the demo. The first transmission appears to be received as the ascii value of capital C but backwards. However, the subsequent letters that we tried to send were received as many 1s. We weren't sure why some transmissions resulted in an 8 digit hexadecimal number, while others only resulted in 2 digit hexadecimal numbers.

4.2.2 Simon Game c file

In its prototype, the Simon game c file simply starts the game via a reset signal and printed message. Also, it takes in the data from the key presses and projects them onto the console. Due to time constraints, we also planned to implement more to the software: Simon game.c should also take in and store a random number sequence generated by the random generator the firmware. As the firmware flashes these images onto the VGA display, the serial_com would store this sequence and compare it with the sequence that will be generated by the user. If the sequence is wrong, then send a did_die signal, which decrements the lives counter. Moreover, there would have been an eventually game limit. If the user reached round 10, then a victory signal would be sent out, both ending the game and congratulating the winner in the console.

5. Presentation and Results

5.1 Testing

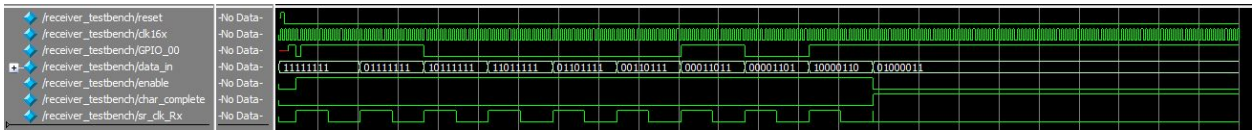
5.1.1 Lab 4

Each of the C programs drawn from the tutorial were tested on the microprocessor we built. Building our microprocessor and running count binary, lights and switches, hello world small were done entirely from the tutorials. Therefore to test them, double check that all steps outlined in the tutorial were followed correctly and check to see that the outputs to the console match that of those mentioned in the tutorial.

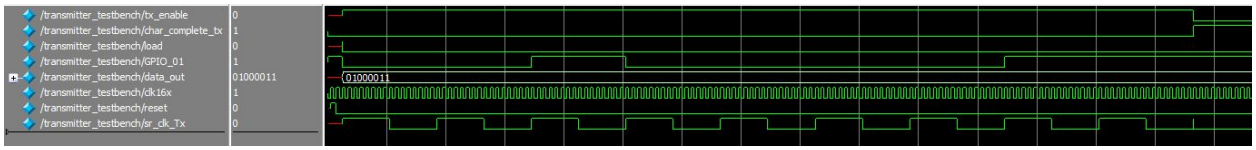
5.1.2 Lab 5 - Serial Communication

The following are modelsim screenshots for the receiver and transmitter in the design.

Receiver Simulation:



Transmitter Simulation:

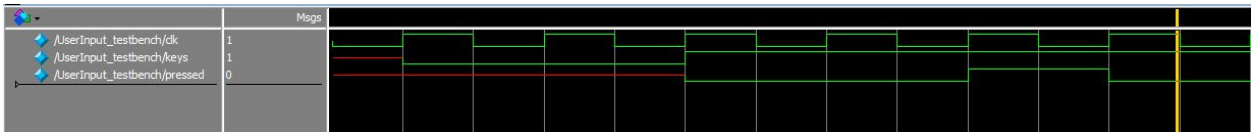


5.1.3 Lab 5 - Extra Credit

The following are modelsim screenshots for the lives to hex and user input in the design.

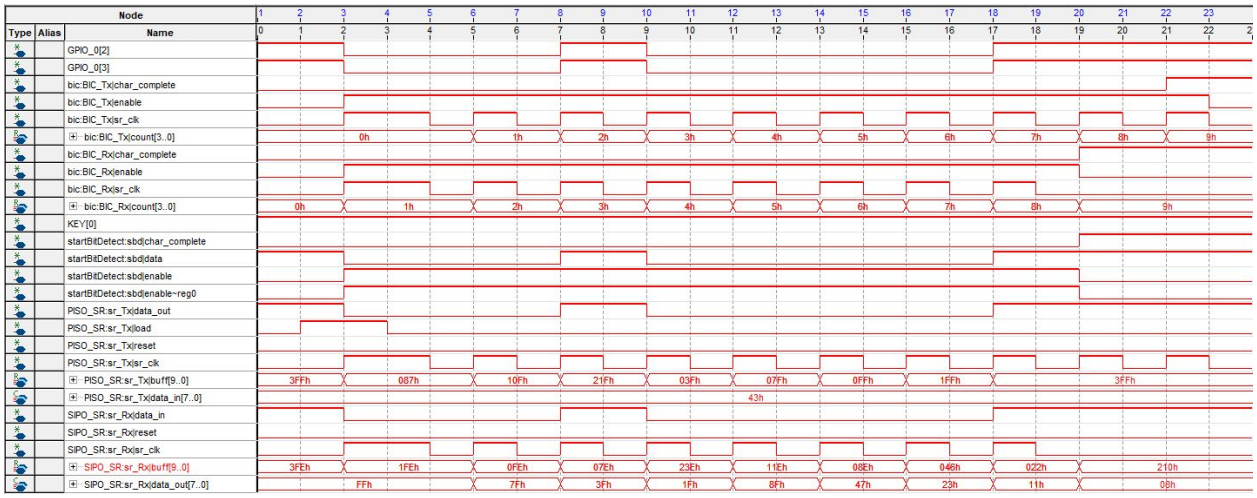
Lives to Hex

User Input



5.2 Results

N.B.: These SignalTap captures were taken with the storage qualifier type set to “transitional.” This eliminates the implemented delays in the circuit in the interest of horizontal space. For timing of the delays, see the Modelsim screenshots above.



This is a signaltap waveform for our serial system.

We do not have any presentables for the Simon Game since it was not fully finished.

6. Error / Failure Analysis

In both labs, we had experienced many problems with uploading our C code (for both labs)

up onto the NIOS II processor. Eclipse gave us many errors when we tried to upload the given code onto our NIOS II processor, often to do with the .elf file (broken or was not existent). To help solve this error, we had to clean and rebuild our project several times; sometimes this method did not work, so we had to build a completely new project and copy over our C program. In the end, we realized that when we uploaded the code onto the NIOS II processor, the reset switch had to be asserted off in order for the code to successfully upload onto the processor.

We also had issues with passing cleaned up pulses from User Input to the Eclipse C programming file. It seems as though one pulse is not enough for the C programming file to register as a "press".

7. Summary and Conclusion

The focus of this project was to take what we had learned in our previous labs and build a complete scanner/base station data collection system with data networking. The entire system was designed to mimic a real world scenario where we had to be able to engineer a complete digital design with limited specification information from the ground up.

Overall, this project gave us the chance to design an advanced digital logic systems in VHDL with a NIOS II microprocessor. We also learned basic networking protocols and applied it to our project; we also learned how to implement a serial based communication network that interacted with other microprocessors. Lastly we learned how to implement a and control from hardware a digital display on the monitor from using a VGA cable and VGA verilog driver.

8. Individual Contribution

Name	Contribution
Arvin Tang	Wrote and tested all .sv files for serial communication program, helped write and debug serial_com.c, contributed to report
Radleigh Ang	Wrote userinput, color, and dealt with the VGA display. Handled nios and c file interfacing. Created the nios processor and dealt with eclipse's incessant crashes. Contributed to report.
Ahmed Moalim	Designed live_to_hex.sv, count_binary.c, hello_world_small.c, random_generator.sv, contributed to report

Signature Page

The undersigned testify to the best of their knowledge that this report and its contents are solely their own work and that any outside references used are cited.

Radleigh Ang

_____

Author 1

Arvin Tang

_____

Author 2

Ahmed Moalim

_____

Author 3