EE 371
Lab 4 & 5
Extending a Simple Microprocessor - Adding Applications With I/O

Name	Student ID	Signature
Justin Allmaras	**1329197**	
Sean Happenny	**1229085**	
Vishesh Sood	**1239599**	

TABLE OF CONTENTS

- Extending a Simple Microprocessor - Adding Applications With I/O
  - TABLE OF CONTENTS
  - ABSTRACT
  - DISCUSSION OF THE LAB
    - Design Specification
    - Design Procedure
    - System Description
    - Software Implementation
      - Battleship Console
      - Wireless Communication
    - Hardware Implementation
      - Nios II Processor
      - Led Matrix
      - UART Communication
      - Wireless Communication
      - SRAM
  - TEST PLAN
    - UART Testing
    - Battleship Console Testing
    - Wireless Testing
    - LED Matrix Testing
  - TEST SPECIFICATIONS
  - TEST CASES
  - PRESENTATION, DISCUSSION AND ANALYSIS OF THE RESULTS
    - Description of Final System
  - ANALYSIS OF ERRORS
    - UART
    - Wireless Communication
    - LED Matrix
  - SUMMARY
  - CONCLUSION
  - APPENDICES
    - Battleship Design Specification
      - System Description
      - Specification of External Environment
      - System Input And Output Specification
      - User Interface

- System Functional Specification
- Operating Specifications
- Reliability and Safety Specification

Battleship Requirements Documentation

- Abstract
- Introduction
- Inputs
- Outputs
- Major Functions

C Code

- LED Matrix

Verilog Modules

- SRAM
- LED Matrix
- Serial Communications

Wireless Communication

Who Did What

ABSTRACT

In these labs, we were asked to extend our microprocessor from lab 3, by firstly adding input and output capabilities, including communication between two separate systems, and then adding our own application to extend the features of the processor. We used the core components from lab 3, such as the SRAM, to successfully allow communication between two systems, and then create a game to play between the same systems.

INTRODUCTION

These labs require us to be able to allow communication between two systems, and then extend an application onto our own Nios II processor from lab 3. In this lab report, we will discuss both labs in detail, and explain how we designed, tested and implemented key functions on our systems.

In the first part, our goal was to allow communication between two systems using wires for input and output. We looked to incorporate basic communication capability, in the same way a bathysphere would communicate with the habitat system in the earlier labs. To do so, we aimed to develop and test an asynchronous serial network implementation on the DE1-Soc board.

In the second part, we aimed to carry on from our success from part one, and implement an application for our Nios II processor. We decided to create a two player game wirelessly with our partner group, and after some thinking, we decided to implement a wireless Battleship game.

In both labs, we used similar tools as before to ensure a working system. We looked to extend our knowledge of tools like Qsys and iVerilog for designing hardware simulations that we ran on gtkwave and Signal Tap. We also look to extend our knowledge in C programming, as well as introduce wireless programming on Arduino to allow for wireless communication between systems.

DISCUSSION OF THE LAB

Design Specification

Lab four required us to extend the properties on our microprocessor by incorporating basic communications capabilities, essentially allowing our bathysphere to communicate with other systems. Our aim for the lab was to be able to develop and test an asynchronous serial network. To do this, we had to figure out the components required to send and receive data. First of all, we accepted the fact that there would be different times at which data could be sent and received. The system could expect a character every second or even every hour. To take care of this issue, we ensured that system knows when data is being sent in order for it to receive it, and also know exactly what each data bit represents, including the start bit, end bit and parity bit. We accomplished this using a procedure known as framing. The start and stop bit are designated to frame the character being sent. By doing so, we can ensure data is grouped into frames which also includes the data and checking (or parity) bit. Our specification required that within the microprocessor, we deal with data in parallel transmission rather than serial. To perform the conversion, we ensured the system knew which part of the frame was being looked at, therefore we know where the data is located and where the parity bit is. Also, it was vital for us to figure out the best timing for our system clocks. To be able to communicate with other systems, we had to ensure they spoke the same language. To do this, we had a mutual agreement with our partner group to use a clock of 9600 Hz. Our specification required us to be able to design and develop a system that relied on a Nios II processor and consisted of a serial-parallel-serial network to support asynchronous message exchanges. To do so, we had to also design, test and implement a console based control system to send and display received data using the C programming language.

Our application had a simple design specification, in which we looked to recreate and implement the classic Battleship game onto our DE1-Soc board with wireless communication. Our aim was to allow two users with access to the system to be able to play a game of battleships with one another, without having to connect any wires to each other's boards. To achieve this, we had to make use of all the tools and expertise we had from our previous labs, such as our skills in C programming, our SRAM implementation along with our Nios II processor, and even some additional knowledge in wireless communication (in our case, using Arduino to transmit and receive data through radio frequency communication.)

Design Procedure

To implement our network interface in lab four, we used the high level block diagram presented to us to break down the system into several hardware modules and processor requirements. Figure 1 shows the high level block diagram for our network interface. Using this diagram, we were able to understand which hardware modules were required, and the roles our microprocessor had to perform to successfully communicate with another system.

We started by designing the core components of the microprocessor, ensuring it performed the tasks as expected by the hardware. We had to ensure it was able to accept characters, load data into shift registers and also trigger data transmission and the counting modules. We required two counters to keep track of both incoming and outgoing data, and these were the bit identifier count (BIC) and the bit sampling count (BSC).

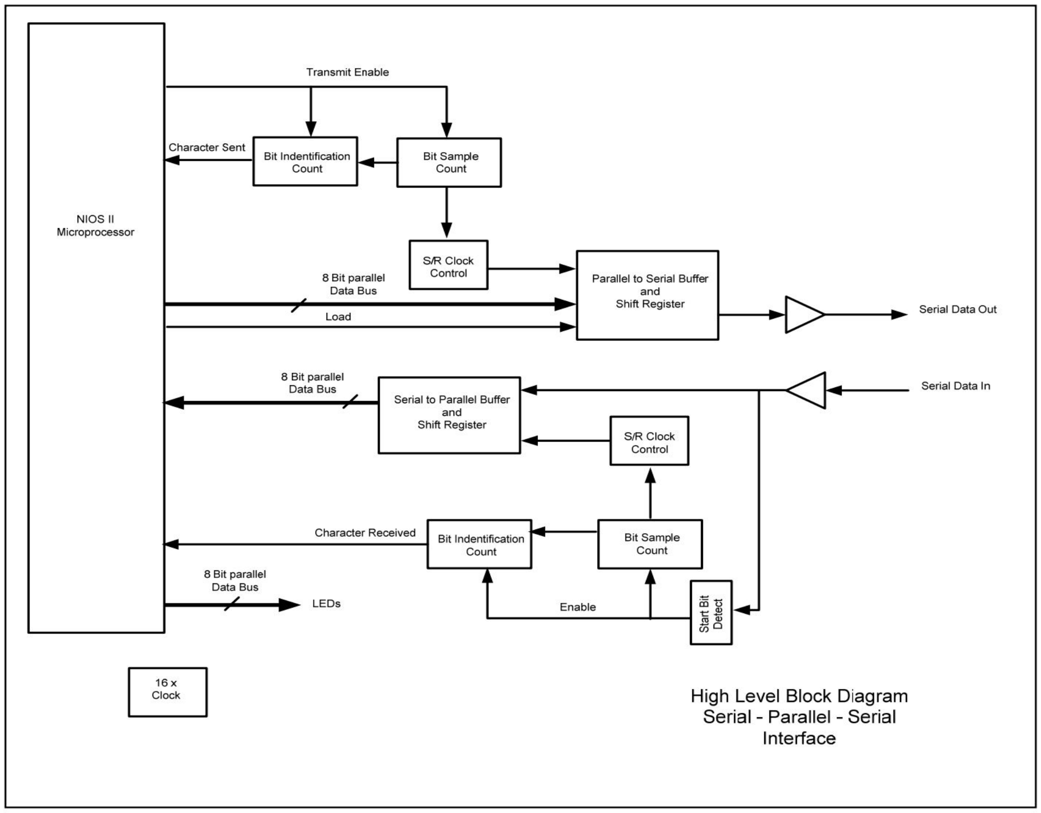


Figure 1: Figure showing our high level block diagram for our network interface.

We used our shift register to propagate the value available at the input on each clock cycle, allowing the bit to be sampled at the halfway point. During the transmit, the shift register takes the entire frame that has been stored and shifted it out serially on each transition of the clock. We used Signal Tap to ensure the functionality was correct and everything worked as expected.

Once we were sure that the hardware was functioning as expected (by using Signal Tap), we worked on our console application, allowing the Nios processor to manage the communication and exchange of data with another system. We ensured that the console first allowed transmission of 1 byte of data (essentially once character), and for the following lab we edited the program to ensure a string can be transmitted without problems.

After successfully demonstrating our working lab 4 system, we decided to work with our other group partners and work on different components of the application, mainly software and hardware implementation. We first agreed on what we wanted to implement in terms of the game. It was a unanimous decision to create a battleship application, and then ensure our SRAM worked and was able to store our board data. While we worked on the SRAM, the other group worked on the program console, ensuring that the user was able to enter data correctly and display the correct information, as well as read data from the SRAM. Then we all focused on getting the console to work flawlessly, and at the same time implemented the wireless communication system. Once those two were complete, we implemented the LED array to read data from the SRAM and light up LEDs on the array accordingly.

System Description

The UART system has the following inputs and outputs:
Inputs

- Parallel Data from the Nios
- Serial Data in from the other board
- 50 MHz clk, which gets divided down to produce a 153.6KHz clk
- Load signal from the Nios to load data to send
- Transmit Enable from the Nios to transmit the loaded data

Outputs

- Parallel data to the Nios
- Char sent and Char received control signals
- Serial data out to other board

The overall battleship system has the following inputs and outputs from the DE1 to the peripherals:
Inputs

- Serial data from Arduino
- Data from console

Outputs

- Serial data to Arduino
- Data to console
- Data to LED Matrix

In the battleship system, the Nios has the following inputs and outputs:
Inputs:

- Input from console
- Parallel data in from UART
- Char send and Char received signals from UART
- 50MHz clk

Outputs:

- Parallel data to UART

- Load and Transmit Signals
- SRAM control signals (chipSelect , readnWrite , outputEnable)
- LED Matrix control signals

Inouts

- Parallel data from SRAM

Software Implementation

Battleship Console

Our sole software implementation was the console for the game, and this was developed in C. The program carries out several tasks, such as taking user input, validating and verifying the data entered, and also displaying a lot of graphical and text based content to the user.

Firstly, in our main function, we print an Ascii art image of the classic Battleship game logo. Following that, we allow the user to set up their ships. Based on what sizes of ships the user wishes to play, the program asks for one coordinate, and then a character (v or h) to determine whether the user wants to align the ship vertically or horizontally. If they enter ‘v’, it will use the coordinate as the topmost coordinate of the ship, and place it on the board. If they chose ‘h’ then the entered coordinate would be the left-most coordinate of the ship. The program ensures that the coordinate entered is not overlapping any other ship, or is not off the board. If that is the case, the program prints “Sorry, that location is off the map or already taken”. It does this by reading the SRAM and finding out if the location (or address) is not already taken. If the check passes, the program writes the data (coordinates and size as a byte) to the same address. It then prints the updated board onto the console. It also updates the LED array by lighting up the LEDs where the ship has been placed.

After the process of setting up the ships is complete, the user is then prompted to enter whether they are player one or player two. Based on the user input, the program either goes into a waiting stage (if the user entered player two), or into a waiting stage in which it prompts the user for a coordinate to fire a missile at. If the user entered player two, the program waits to receive a coordinate through the wireless transmission. Once player one enters a coordinate, the data is sent to the opponent’s board, and the other board returns whether the user had a hit or a miss. Based on that, the console updates the board for player one, and then asks the player to wait as it is now player two’s turn to fire a missile. On the other console, the user who just accepted the missile is shown their updated board based on whether the opponent hit or missed their ships, and also is prompted to enter a coordinate to fire at.

There is a lot of background processes taking place once a missile is fired. Firstly, the data is sent through wireless communication and is checked on the opponent's board whether it was hit or missed. The program then translates the data returned from the other board, and has to print a message accordingly. The same process is repeated but for two bytes when the user sends a coordinate.

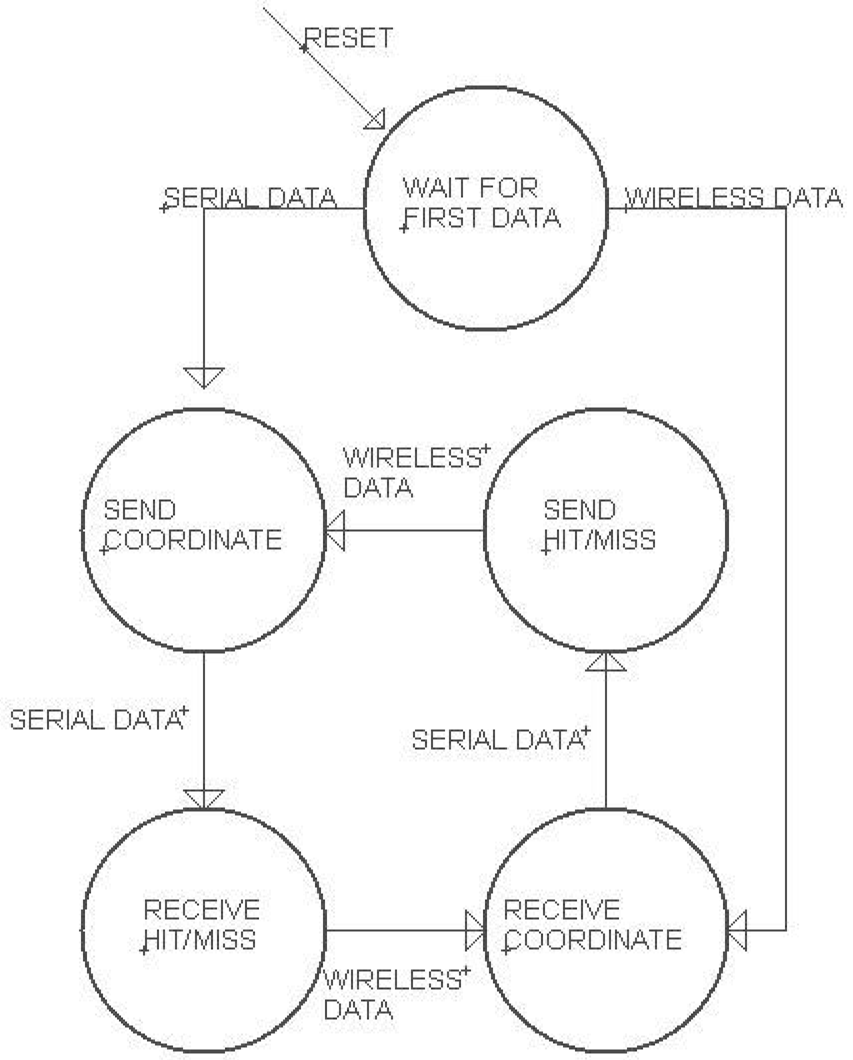
This process is repeated until either the console is shut down, or one user has completely sunk the opponent's ships. In this case, the console displays by comparing the total hits possible with the number of hits made by the user, to determine who won and print an output to the console.

Wireless Communication

The wireless communication system was programmed using the Arduino IDE. We had to ensure both chips were programmed with correct receive and target transmit addresses, so we identified each board by a board number as a global variable. After that, we identified the two chip select and chip enable pins (in our case, pins 9 and 10), and initialized the data array to store the sent and received coordinates. Lastly, the wireless modules are initialized to prepare for the transmission or reception of data.

Both wireless boards then enter a state that waits for either wireless data or serial data. If a board receives serial data, then it enters the send state and sends the coordinate it received from the DE1. The other board see that wireless data is ready and goes into the receive state to receive the wireless data. This system works on the assumption that only one DE1 will send serial data at the beginning of the game and this is supported in the game by one person selecting player 1 and one person selecting player 2.

We had a function that would first send a coordinate, receive a coordinate, send a hit/miss response, and receive a hit/miss response. The hit miss package, whether it was a hit or a miss, is a 2 byte value, the first byte being the character 1 or 0 for hit or miss respectively, and then a byte for a null character, so that the system receiving knows that the data transmission is over. We had a similar setup for the send and receive coordinate functions, in which we use a three byte package to send the two bytes of the character and a null terminator to signal the end of the transmission. These functions call each other to maintain the cycle of the state machine, which can be seen in Figure 2.



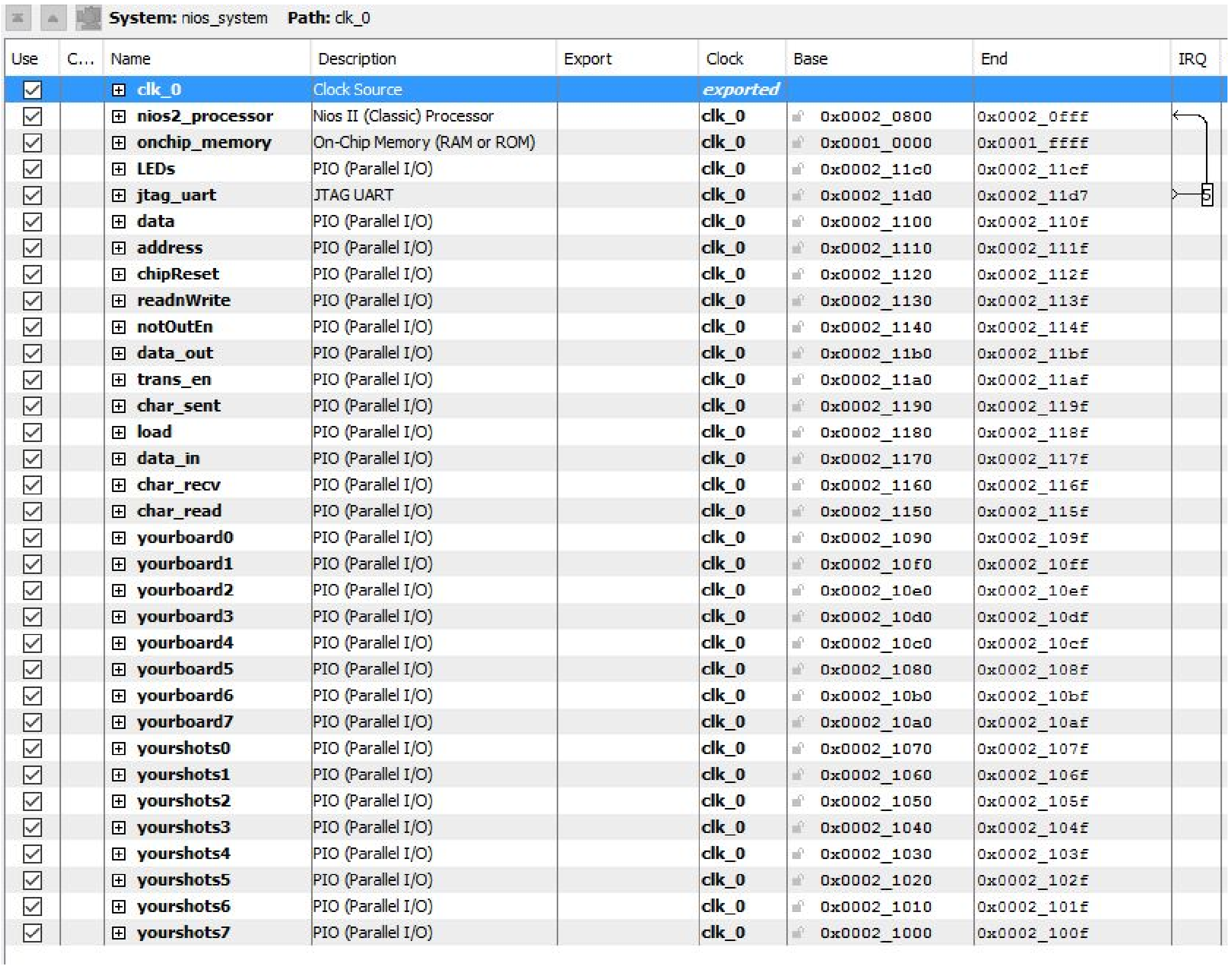
.

Figure 2: Software state machine for wireless communication

Hardware Implementation

Nios II Processor

Our NIOS II system for this project was built off of the NIOS system for our previous project, which also included an SRAM and serial module. We added 16 parallel I/O outputs to send the location of our ships and the location of our shots fired from the C program to the LED Matrix. We also had 13 other PIO ports to send and receive control signals and serial data to and from our hardware modules. This allowed us to transfer the information needed to monitor and store the state of the game, send and receive shots, and process and react to the user input. Below is a screenshot of our fully configured NIOS II system.



System: nios_system Path: clk_0

Use	C...	Name	Description	Export	Clock	Base	End	IRQ
✓		⊞ clk_0	Clock Source		*exported*			
✓		⊞ nios2_processor	Nios II (Classic) Processor		clk_0	0x0002_0800	0x0002_0fff	
✓		⊞ onchip_memory	On-Chip Memory (RAM or ROM)		clk_0	0x0001_0000	0x0001_ffff	
✓		⊞ LEDs	PIO (Parallel I/O)		clk_0	0x0002_11c0	0x0002_11cf	
✓		⊞ jtag_uart	JTAG UART		clk_0	0x0002_11d0	0x0002_11d7	5
✓		⊞ data	PIO (Parallel I/O)		clk_0	0x0002_1100	0x0002_110f	
✓		⊞ address	PIO (Parallel I/O)		clk_0	0x0002_1110	0x0002_111f	
✓		⊞ chipReset	PIO (Parallel I/O)		clk_0	0x0002_1120	0x0002_112f	
✓		⊞ readnWrite	PIO (Parallel I/O)		clk_0	0x0002_1130	0x0002_113f	
✓		⊞ notOutEn	PIO (Parallel I/O)		clk_0	0x0002_1140	0x0002_114f	
✓		⊞ data_out	PIO (Parallel I/O)		clk_0	0x0002_11b0	0x0002_11bf	
✓		⊞ trans_en	PIO (Parallel I/O)		clk_0	0x0002_11a0	0x0002_11af	
✓		⊞ char_sent	PIO (Parallel I/O)		clk_0	0x0002_1190	0x0002_119f	
✓		⊞ load	PIO (Parallel I/O)		clk_0	0x0002_1180	0x0002_118f	
✓		⊞ data_in	PIO (Parallel I/O)		clk_0	0x0002_1170	0x0002_117f	
✓		⊞ char_recv	PIO (Parallel I/O)		clk_0	0x0002_1160	0x0002_116f	
✓		⊞ char_read	PIO (Parallel I/O)		clk_0	0x0002_1150	0x0002_115f	
✓		⊞ yourboard0	PIO (Parallel I/O)		clk_0	0x0002_1090	0x0002_109f	
✓		⊞ yourboard1	PIO (Parallel I/O)		clk_0	0x0002_10f0	0x0002_10ff	
✓		⊞ yourboard2	PIO (Parallel I/O)		clk_0	0x0002_10e0	0x0002_10ef	
✓		⊞ yourboard3	PIO (Parallel I/O)		clk_0	0x0002_10d0	0x0002_10df	
✓		⊞ yourboard4	PIO (Parallel I/O)		clk_0	0x0002_10c0	0x0002_10cf	
✓		⊞ yourboard5	PIO (Parallel I/O)		clk_0	0x0002_1080	0x0002_108f	
✓		⊞ yourboard6	PIO (Parallel I/O)		clk_0	0x0002_10b0	0x0002_10bf	
✓		⊞ yourboard7	PIO (Parallel I/O)		clk_0	0x0002_10a0	0x0002_10af	
✓		⊞ yourshots0	PIO (Parallel I/O)		clk_0	0x0002_1070	0x0002_107f	
✓		⊞ yourshots1	PIO (Parallel I/O)		clk_0	0x0002_1060	0x0002_106f	
✓		⊞ yourshots2	PIO (Parallel I/O)		clk_0	0x0002_1050	0x0002_105f	
✓		⊞ yourshots3	PIO (Parallel I/O)		clk_0	0x0002_1040	0x0002_104f	
✓		⊞ yourshots4	PIO (Parallel I/O)		clk_0	0x0002_1030	0x0002_103f	
✓		⊞ yourshots5	PIO (Parallel I/O)		clk_0	0x0002_1020	0x0002_102f	
✓		⊞ yourshots6	PIO (Parallel I/O)		clk_0	0x0002_1010	0x0002_101f	
✓		⊞ yourshots7	PIO (Parallel I/O)		clk_0	0x0002_1000	0x0002_100f	

Figure 3. Our complete NOIS II system, showing the clock, processor, memory, UART, and all of the parallel I/O ports.

Led Matrix

We used an 8 by 8 LED Matrix to show the status of our game board to the player. The LED Matrix was the same size as the game board, so each LED corresponded to a space on the game board. The LED Matrix also had green and red LEDs, so we were able to use the two colors to represent different game board states. We used the green LEDs to show where the player's ships were and with the push of a button, the player could toggle to the red LEDs which showed where the player has fired. The driver for the LED matrix is implemented in verilog and accomplished through the use of 24 GPIO pins. The 24 GPIO pins are able to drive 128 LEDs through the use of rapid switching through all of the rows of LEDs so that the human eye cannot distinguish the switching. We used a clock divided 15 times from 50MHz to cycle through the rows. The result is LEDs that appears to be constantly on or off.

UART Communication

We developed our UART communication module in Lab 4, implementing it by using several submodules to handle the detection of a character, receiving a character, and transmitting a character. The detection of a character is done in a startBitDetect module, which constantly monitors the serial receive line for a transition from the quiescent high state to a low state (the start bit is always a 0). Receiving a character is done by using a serial to parallel shift register, a clock that counts 10 bits, and another clock that ensures we sample the bit at approximately its midpoint to avoid any effects of clock skew or hold time violations. Once 10 bits have entered the register, a signal is sent high to indicate that a character has been received and the character can be read from the register via 8 parallel signal lines for the 7 data bits and 1 parity bit. Sending a character is basically the reverse of receiving: 8 parallel lines load the data and parity bits into a parallel to serial shift register; a low start bit is put onto the serial transmit line, followed by the 8 message bits, and the high end bit. One bit is sent out per clock pulse, which is derived from the system 50MHz clock to be 9600Hz.

Wireless Communication

We implemented wireless using a custom PCB with an ATMEGA328 microcontroller to give it Arduino functionality. We burned the ATMEGA with an 8MHz bootloader and powered it off of 3.3V to match with both the DE1 GPIO voltage and the wireless module IO voltage. The wireless module is an off the shelf 2.4GHz RF module. The module is centered around the nRF24L01 transceiver integrated circuit and communicates to the Arduino through an SPI data bus. The ATMEGA chip communicates with the DE1 using the UART communication protocol from lab 4. The PCB had two LEDs that we used for debugging and display purposes. There was an orange LED that was tied to the chip select line of the SPI bus and there was a blue LED attached to a GPIO and we used this LED to show when the board was waiting for serial data.

SRAM

For this lab, we implemented a fixed version of our SRAM from lab 3. We used this to store our game boards for our shots and the opponents shots. This was implemented by using the SRAM to store 8 bytes for our shots and 8 bytes for the opponent's shots. This module contains a 2048x8bit array to store our data on the positive edge of signal readnWrite (low when we want to read). This loads the data from the data line into the array at the address from the address line when the reset signal chipSelect is low. To read from the SRAM, we set the signals notOutEnable and chipSelect low, which assigns the values stored in the array at the address on the address line to the data line.

TEST PLAN

Our test plan for this project was to test each major component individually and combine them one at a time, ensuring each addition is fully functional. Our major components include the SRAM, wired serial hardware, wireless serial hardware and software, the LED Matrix hardware and software, and the user interface/game logic software.

UART Testing

Before testing we did simple simulations of some of the submodules using iVerilog and GTKwave to verify the correct functionality. This actually introduced some issues because iVerilog allows use of statements that cannot be synthesized, so our initial version of the verilog code was not synthesizable, and when we changed the code, we made some simple mistakes that caused the system to fail. Using signaltap we were able to see what was going wrong and fix the issues in the synthesizable verilog.

We began testing the serial communication by hooking up the transmit and receive lines to attempt to communicate with ourselves by making sure that what we sent is what we received. We did this by simply connecting a wire between the two GPIO pins on the DE1 breadboard. Once we got this working, we tested communicating with another group by connecting our transmit to their receive and our receive to their transmit and we made sure to connect both of our grounds to ensure a common reference. We tested this communication scheme considerably to make sure that the system was robust and produced correct results.

Battleship Console Testing

Our partner group handled most of the console development and testing, but they tested small segments of the code as they developed to give the final version the best chance of working. Once the first version of the code was created they tested the most basic scenario of the game ignoring most of the hardware peripherals (wireless , led matrix). This allowed them to fix all of the bugs and make sure that the control flow through the code was correct. This also allowed for error handling testing to make sure that bad user input couldn't break the console execution. Once all of the errors and bugs were fixed, we began to implement the extra hardware peripherals one by one to give the easiest time of debugging a specific system if something did not work.

Wireless Testing

The wireless system underwent many different stages of testing to work out the interface with the DE1 and to test and debug its functionality. We first connected both wireless modules to computers and used the Arduino IDE serial monitor to view the transmissions through the wireless module connected to the computer. We also used an oscilloscope in this stage to verify that the output wave turned out as expected. We also used this as a test of the state machine for the wireless communication.

After finishing testing with the two computers, we connected one of the wireless systems to a DE1 and kept the other connected to the computer. We used this configuration to test sending from the DE1 to the Arduino and also receiving from the Arduino. We did most debugging in this stage because we were able to go through all of the stages of a game of battleship and see what was being sent from the DE1, so it allowed us to correct all of our problems.

The last stage was to use both DE1 boards and test the wireless through multiple games of battleship. This was pretty easy since we had already verified functionality from the computer/DE1 test, so it went smoothly. We also did some basic range testing to make sure that we would not drop packets at a distance of about ten feet, and we had 100% success at this range. We did not try to find the distance at which the modules would fail since we planned to operate the boards within five feet of each other.

LED Matrix Testing

We began by using simple verilog behavioral code to make sure the driver was working properly and that all of the individual LEDs were still functioning properly. This also tested to make sure that the wiring was done correctly and robustly. We had some problems with wires wanting to slip out of the GPIO holes in the breakout, so this was an important step to make sure that our wires were properly connected.

We then went on to test whether or not we could write from the SRAM to the LED matrix. We tested doing this in hardware, but ran into problems, so we changed our approach and tested whether or not we could write to the LED matrix from the NIOS, and we found out that we could accomplish this, so this became our final approach.

TEST SPECIFICATIONS

The test specification for our Battleship game system is to verify that the user input is handled correctly in C, the game logic is correct, data is correctly sent to and received by the other game board, and the game state is correctly updated in the SRAM and on the LED matrix. We will do this by running the C program in debug mode and examine the variables and parameters of each function to ensure that they are correct while also monitoring the game logic to check that it is impossible to enter any illegal game states. Additionally, we will use the C debugger to monitor the data received on the serial input line to make sure that the necessary interboard data is being transmitted correctly and according to our specifications. We will also test the wireless system separately by using a board to send/receive data from its wireless radio to a radio connected to a laptop to view the received/sent data and verify it is correct. Further, we will use the C debugger to check that the data read from the SRAM is correct and verify that the SRAM data is correct on the LED matrix display.

TEST CASES

For the C program, we need to test that only valid user input is accepted, the game logic limits the actions the user can perform and the states the game can enter, data is communicated correctly with the opponent's board, and the game state is saved to the SRAM and displayed on the LED matrix correctly. We will test the acceptance of only valid user input by providing both valid and invalid user input to the console--for example, an uppercase letter between A-H followed by a number between 0-8 is valid input for a game coordinate, while a lowercase letter followed by a number is not. We will test the game logic by repeatedly playing the game and trying to enter an illegal state at every state transition--for example, by entering illegal inputs or entering inputs when the game is not expecting user input. We will test the serial and wireless communication by repeatedly playing games with another board and verifying that data is correctly sent and received on each board using the Eclipse C debugging and just running the game. Similarly, we will test the SRAM and LED matrix by playing games under the C debugger and simply running them and verifying that variable values and the ships/shots displayed on the LED match what the current game state should be.

For the communication modules, we need to verify that the wired UART and wireless radio modules work correctly separately and combined. We will test the wired UART by connecting a wire between our transmit and receive lines so that we receive the same data we transmit and can thus verify the functionality of both at the same time. To do so, we will send various ASCII characters and ensure that the transmitted data meets the specifications--namely: 10 bits are sent, 1 start bit (low), 7 data bits, 1 parity bit, and 1 end bit (high); the baud rate of 9600 bits/s is maintained; and our parity generation and verification is correct. We will begin testing the wireless communications module by connecting one radio to the transmit and receive lines of our board and another radio to a laptop running software that allows us to send and receive arbitrary data. We will send ASCII characters over this link and verify that they are sent and received correctly, with the proper bit order/endianness and correct parity.

For the SRAM, we had already verified its functionality in Lab 3, so we will simply repeat the same test cases here to ensure that integrating it with the other modules did not compromise its functionality. We will test both operations, reading and writing. The SRAM needs to be able to read and write to all 2048 memory addresses. Testing each one would be unreasonable, so it is important to test a few different cases from different areas of the memory. It is also important to test the edge cases to make sure that they are covered properly. This means that it is important to test reading and writing from addresses 0 and 2047 as well as a few in the middle. It is also important to make sure that there is no possibility for a read and write action to occur at the same time, because that will most likely produce damaging results as two sources will be fighting over control of the same data lines.

For the LED matrix, we need to test that it properly displays both the green and red LED signals and switches between the two with a keypress. We will test the green and red driver signals by first assigning hardcoded values to the driver registers and verifying that the LED displays the hardcoded values. Then, we will assign variable values from the SRAM to the LED drivers to ensure that the module can display input data as well. In both cases, we will test the keypress function to switch between the green and red LED drivers.

PRESENTATION, DISCUSSION AND ANALYSIS OF THE RESULTS

Overall, our system functioned exactly as designed, met all of the design requirements, and included several additional features--the wireless communication, LED matrix, and external communication on/off “Battleship Engagement” toggle switch. Each piece of our system functioned as designed. We were able to read and write our ship and shot boards to and from the SRAM without any difficulties. We had previously implemented the SRAM incompletely in Lab 3, but were able to get it working for this project. A screenshot of our Signal Tap analysis for the SRAM module is included below, showing the GPIO input and output (physRx and physTx), our control signals (transmitEnable, load, charSent, and charReceived), the parallel data lines to and from the NIOS II system, the counter registers and clocks for bit detection and word detection, and the serial data in and out registers that hold our data before being loaded onto the output GPIO line.

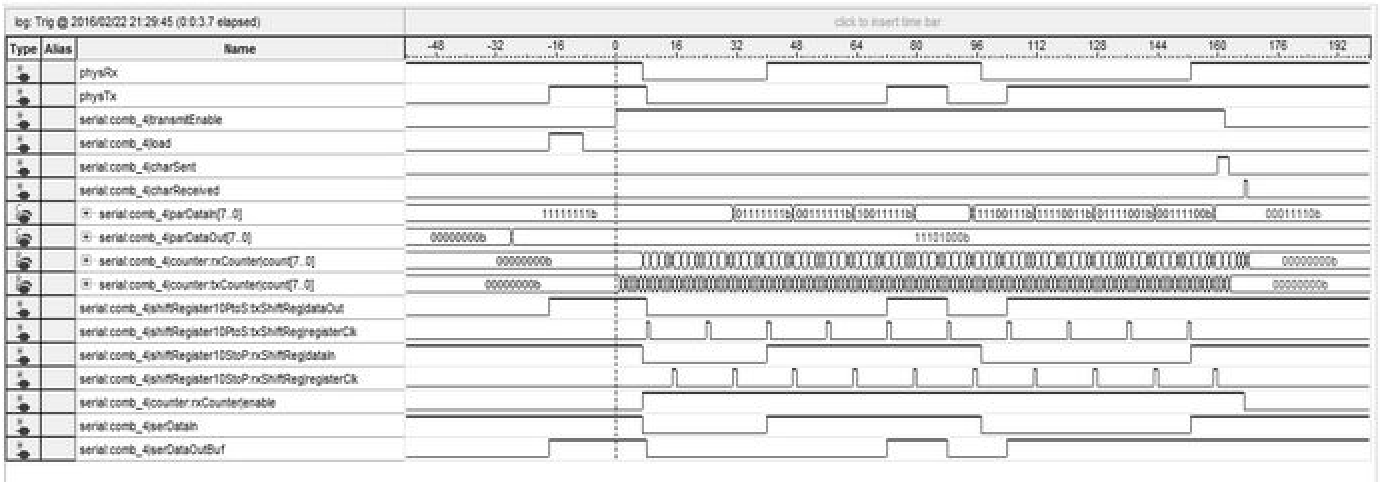

log: Trig @ 2016/02/22 21:29:45 (0:0:3.7 elapsed)
click to insert time bar
Type
Alias
Name
-48
-32
-16
0
16
32
48
64
80
96
112
128
144
160
176
192
physRx
physTx
serial:comb_4|transmitEnable
serial:comb_4|load
serial:comb_4|charSent
serial:comb_4|charReceived
serial:comb_4|parDataIn[7..0]
serial:comb_4|parDataOut[7..0]
serial:comb_4|counter:rxCounter|count[7..0]
serial:comb_4|counter:txCounter|count[7..0]
serial:comb_4|shiftRegister10PtoS:txShiftReg|dataOut
serial:comb_4|shiftRegister10PtoS:txShiftReg|registerClk
serial:comb_4|shiftRegister10StoP:rxShiftReg|dataIn
serial:comb_4|shiftRegister10StoP:rxShiftReg|registerClk
serial:comb_4|counter:rxCounter|enable
serial:comb_4|serDataIn
serial:comb_4|serDataOutBuf
11111111b
01111111b
00111111b
10011111b
11100111b
11110011b
01111001b
00111100b
00011110b
00000000b
11101000b

Figure 4. Signal Tap output for our SRAM module.

While our LED matrix had some bugs during development, we were able to resolve all of them and deliver a fully functional 8x8 bi-color LED display to show the player's current ship layout and damage taken and, with a key press, switch to displaying the shots they've taken on their opponent. This was implemented nearly perfectly, with no flickering or lag between the ship and shot status in the game state and displaying it on the LED matrix. The only issue was that the top row of our LED matrix seemed to be dimmer than the rest of the board, which could be due to a glitch in our row sink driver register. In the Signal Tap output below, a glitch is evident in the rowSink[7] signal, where it is staying high approximately 4 times longer than it should be. This could adversely affect the amount of current that the LEDs on this row receive and could be the cause of their relative dimness. However, every other signal shown below is operating as designed. The greenDriver register is the green LED driver that shows the player's ships; the redDriver register, which shows the player's shots, was not included in this Signal Tap analysis due to recording buffer size constraints.

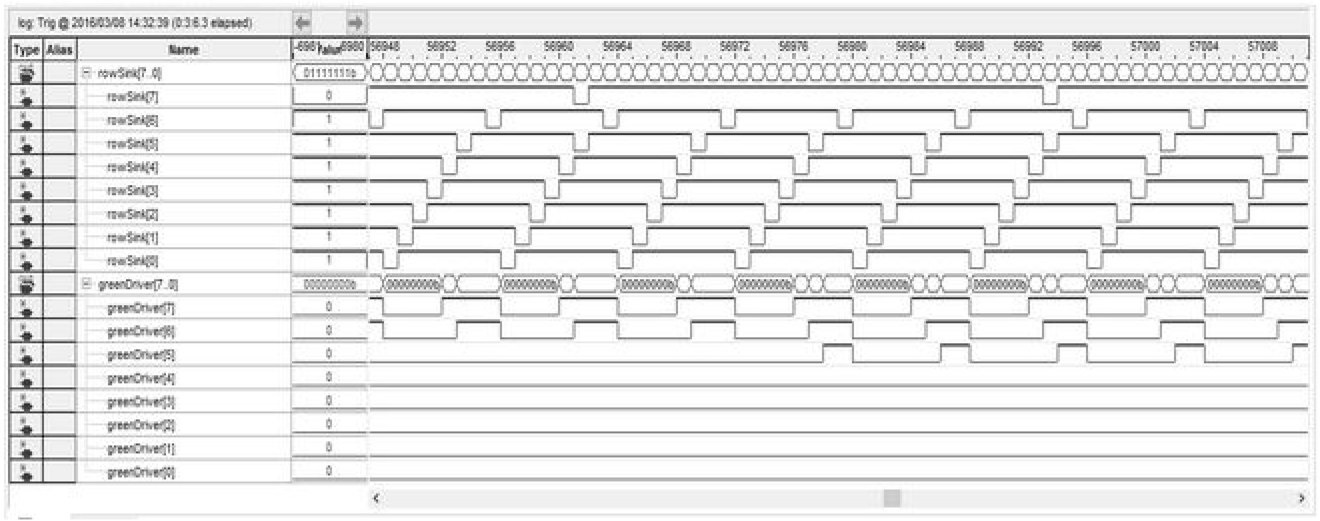
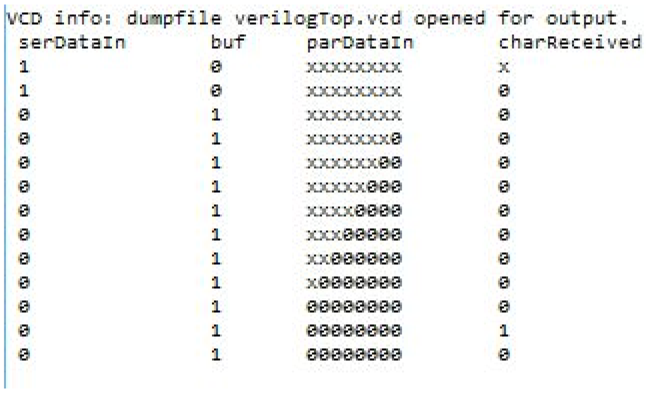


Figure 5. Signal Tap output for our LED matrix driver.

Simulations played an important role in making it to the final design. We primarily used iVerilog for the initial simulations of the communication system and once we were close to a correct implementation, we used signal tap to verify and correct any errors.



```
VCD info: dumpfile verilogTop.vcd opened for output.
 serDataIn        buf      parDataIn       charReceived
 1                0        xxxxxxxx        x
 1                0        xxxxxxxx        0
 0                1        xxxxxxxx        0
 0                1        xxxxxxx0        0
 0                1        xxxxxx00        0
 0                1        xxxxx000        0
 0                1        xxxx0000        0
 0                1        xxx00000        0
 0                1        xx000000        0
 0                1        x0000000        0
 0                1        00000000        0
 0                1        00000000        1
 0                1        00000000        0
```

Figure 6: Test of receiving data

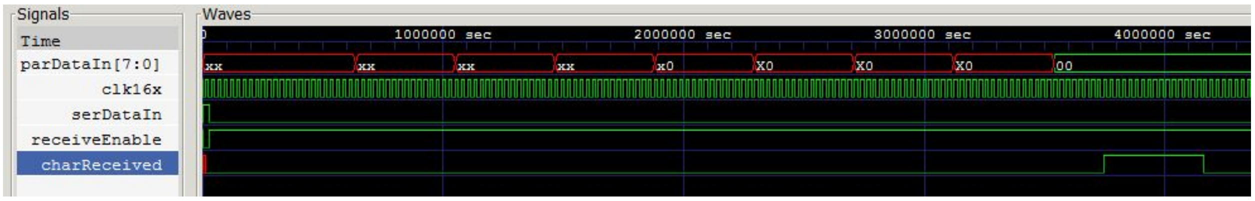


Figure 7: Gtkwave of receiving data

These simple simulations allowed us to move forward with the project and begin testing on the DE1. We did run into a few errors due to iVerilog allowing code that cannot be synthesized, so we had to fix a couple things before we had a fully functional system, but due to simulation we got very close on our first attempt. We used signaltap once we had the system running on the DE1 for a more detailed look at debugging. From here we were able to implement a final design for our UART communications, which played a central role in creating an operational battleship game.

Description of Final System

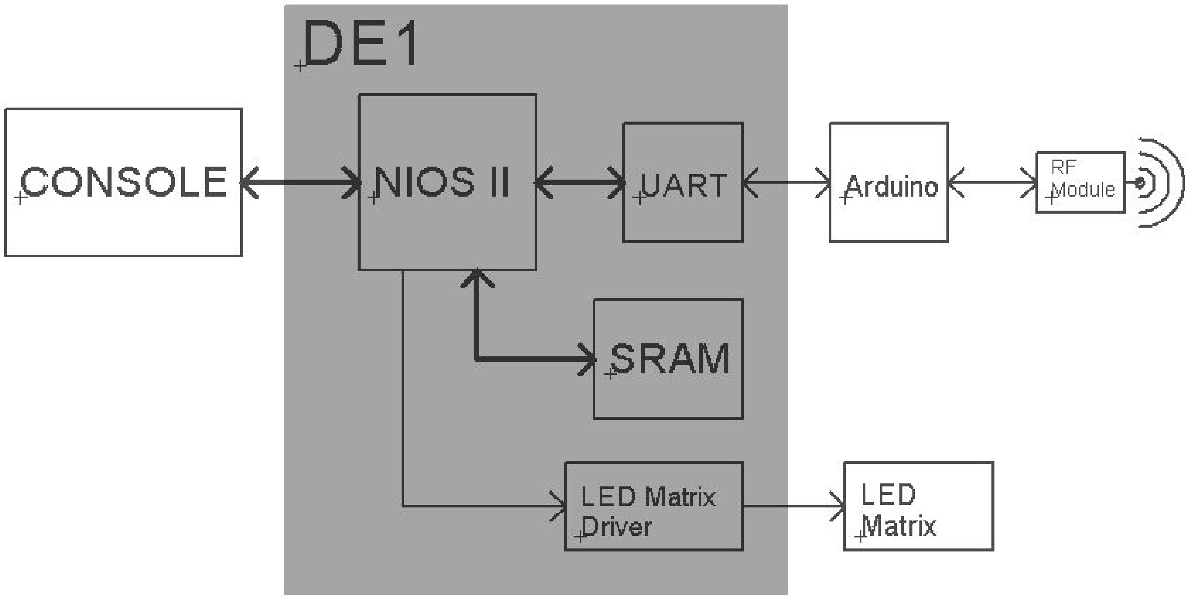


Figure 8: Block Diagram of final design

This block diagram shows the basic structure of the major subsystems in our final Battleship game and how they interact. Each group had one of these setups and they were able to communicate through the wireless module. Although the majority of the system stayed within the DE1, we tried hard to implement additional hardware peripherals to make for a more interesting game of battleship.

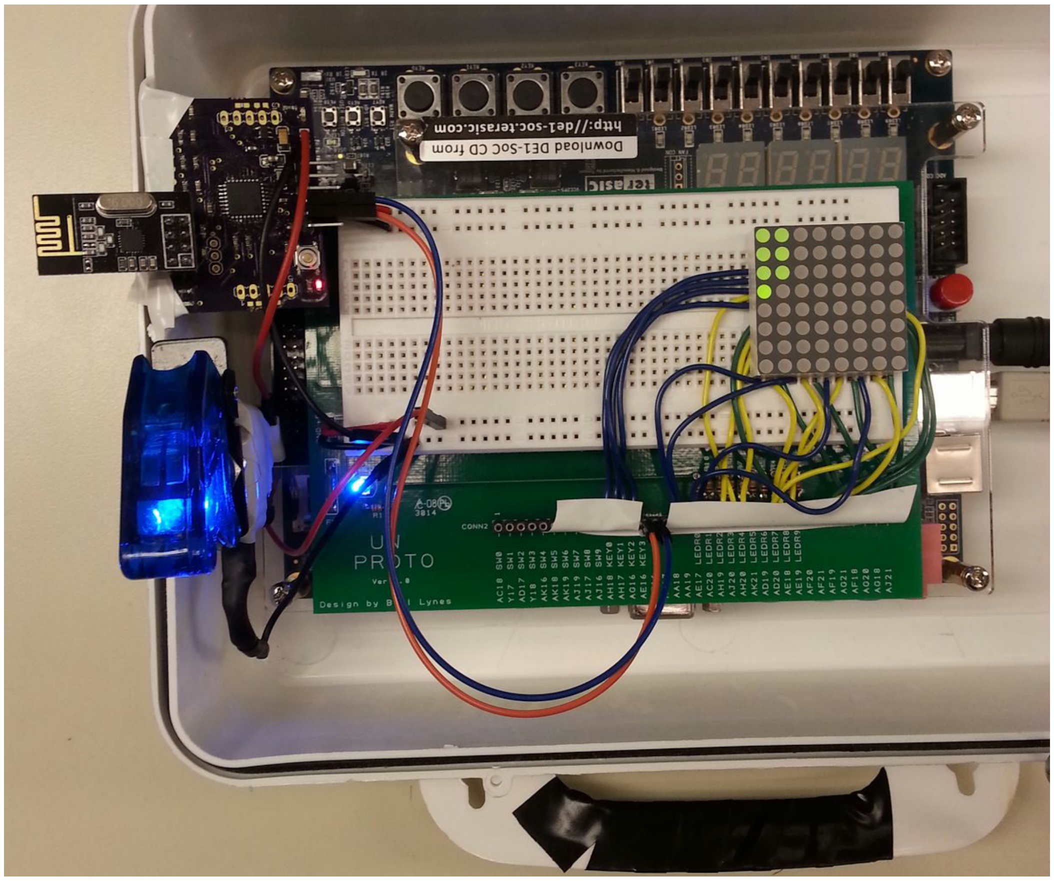


Figure 9: A picture of our final design, showing our ships using the LED array and the wireless transmitter.

Figure 9 clearly shows the addition of extra hardware peripherals. On the right of the board is the LED matrix, which is currently displaying the positions of our ships. If key 1 is pressed, the LED matrix will show where your shots have been fired with the red LEDs. On the top right is the wireless PCB and module. The orange and blue wires are the UART transmit and receive lines. In the lower left is a switch to turn on power to the wireless board. Inspired by the battleship theme, the switch is activated only by first lifting a cover and then flipping the switch, much like a missile launch switch. This command is given to the user to flip the switch in the console code after the user has placed their ships, preparing them for the battle that lies ahead. We also integrated the DE1 board into the carrying case for the board, so just like the classic game of battleship, there was a lid to block the view of any would be cheaters.

ANALYSIS OF ERRORS

UART

While working on the serial communications systems, we discovered by using Signal Tap several errors in our Verilog hardware design affecting the receive and transmit functions of our system. First, we noticed that the receive bit and character counters were always counting, regardless of whether the receive line was quiescent or active. We found that this was due to an incorrect "always" block statement in our start bit detect module, which made the system think a start bit was repeatedly being detected. We fixed this by adding an additional control signal to this "always" block. The second major bug we discovered was in our parallel to serial conversion module, which converts the parallel dataline from the NIOS processor to the serial output for the transmit line and loads the data onto the line. We found that the data was never being loaded onto the transmit line because of an an "always" block that was missing a control signal and an "if/else if" block that was in the wrong order (Verilog complained about it, anyway). Correcting these issues fixed our transmit module and we immediately saw data being loaded onto the transmit line, as well as the receive line as we had a working receive module as well. The last major issue we ran into was that we were sending the data bits in the opposite order from our partner team, so we were not able to communicate properly with them until we reversed our shift registers in our Verilog code. This was a fairly easy problem to detect, as we could tell from the character we received that the data bits were simply reversed. Fixing the Verilog code was also easy, as it only involved putting the received data bits into the lowest index in our shift register and shifting up instead of the highest index and shifting down, and putting the highest index bit onto the transmit line and shifting up instead of the lowest index and shifting down.

Wireless Communication

We encountered several errors while working with the wireless communication system. The first was deciding what data type to use within the arduino to get a 10 bit serial output. There were three options that stored 8 bits of data: char, uint8_t, and byte. Using an oscilloscope we found that using the Serial.print() function of the Arduino produced a 10 bit output for the char type, but uint8_t and byte produced about 20 bits of output data, which we assumed to be a data byte and some sort of type byte. We wanted to use the uint8_t data type because the wireless module had a compatible Arduino library for it. We found that the Serial.write() function produced the 10 bit output that we were looking for.

We also ran into a problem where performing a soft reset of the Arduino would cause the TX line to fall low, which caused the DE1 to think it received serial data. To fix this, we added a pull up resistor on the TX line to keep the quiescent state on a soft reset. A hard reset still caused this problem, but we solved this in software when the Arduino is turned on by the switch.

LED Matrix

We initially tried to integrate the LED matrix control into the SRAM module, whereby the LED matrix would constantly be driven by the memory locations storing the player's ships and shots for the green and red LED driver lines, respectively, with a keypress to toggle between the two. However, we were only reading 0's out of the SRAM, even though these addresses stored nonzero data--we verified this by writing to every location in the SRAM and reading it back in our C program. We assigned hard-coded bit values to the LED drivers, which worked as expected (i.e. the bit patterns showed up correctly on the LED matrix), so it was only reading from the SRAM and assigning this data to the drivers that was not working for some reason. To fix this, we moved our LED matrix driver assignments to our C program by creating an 8-bit PIO in our NIOS II system and directly assigning data values to these PIO addresses. Again however, we faced errors, this time in the form of corrupt memory. Data values that should not be affected by the driver pointers (PIO addresses) were being corrupted somewhere "in between" functions of our C program. This even caused our program to experience an unknown interrupt (according to underlying Altera interrupt error handling code) when exiting one of our functions that had nothing to do with the LED drivers. We attributed this to having used the same source code as our partner group, but not using their NIOS II .qsys file. To fix the LED matrix, we started over with an exact copy of our partner group's C code and .qsys file and re-implemented the LED matrix control in the C code. This solved the errors we were experiencing, and had wasted several hours spent debugging, and left us with a working LED matrix.

SUMMARY

We succeeded in the main objectives of this lab: we successfully communicated via serial interface between two systems; developed an SRAM module to store up to 2048 bytes of data; interfaced with an 8x8 LED matrix to display data from our SRAM module; and built a hardware and software implementation of the game Battleship, communicating over our serial interface, using our SRAM to store the game state, and displaying the game boards on 8x8 LED matrices. In order to develop our serial interface, we successfully built and tested an asynchronous serial interface implementation on the DE1-SoC board. We successfully combined our C programming and Verilog design skills to create a two player game of Battleship played wirelessly with our partner group. Overall, we looked to extend our knowledge of tools like Qsys and iVerilog for designing hardware modules, Quartus for synthesizing them to an FPGA environment, and Eclipse with a custom plugin to enable easy coding for the NIOS II processor on our DE1-SoC board. We also look to extend our knowledge in C programming and challenge ourselves to go above and beyond the call of duty by implementing wireless serial communication between our two boards.

CONCLUSION

To conclude, both labs were major successes, as we were able to get all the required components to work as expected. We were successful in building a fully functional and well tested system that was able to communicate with another system. We were also able to take the system a step ahead by adding an application to the system. We successfully added the game Battleships to our system, and were able to communicate with another system wirelessly.

APPENDICES

Battleship Design Specification

System Description

The classic Battleships game was made for two players aiming is to sink the opponent's ships by firing missiles at coordinates on the opponent's board. Our new Battleships system follows the same specifications as the classic game, but with a touch of technology. We will be using a NIOS processor based embedded system to wirelessly transmit data to fire missiles at opponent's ships. The same system is used to communicate success and failure of the missiles, and determine the winner.

Our Battleships system essentially carries out the same tasks as the classic game, but provides a graphical interface to communicate data from, allowing users to play from further distance than the original game, using wireless communication.

Specification of External Environment

The system works in a variety of climates and environments, however the optimum conditions to play the game is indoors at room temperature. The system is not waterproof, and should not be played near actual oceans and seas. The system weighs around 250 grams and is portable as long as there is a power input to the DE1-SoC board.

System Input And Output Specification

System Inputs

The system must be able to take commands from a player's keyboard, ensuring that the user can enter coordinates in the form of a letter and a number (e.g. A1 or B5). This allows the user to place their ships on the game board. The same input system must be able to set the layout of the ships. The system must also have a wireless receiver that takes data from a transmitter, ensuring data is collected from the opponent's system.

System Outputs

The system ensures that the user knows whether their shot was a hit or a miss, and for this, there must be a console output from the system to the user's PC display. The system is also able to transmit data received from the user's input, using a wireless transmitter. There is an LED matrix array to display the ships and hits/misses.

User Interface

The user interface for the system must be in the form of a console input/output in which the user can see what they need to do, in terms of setting up their ships or firing missiles or notifying if they won. The user does not have to have any prior knowledge of any programming language or anything more than how to use a computer to be able to use the system. Also, the user has visible access to the LED array to see their ship layout, along with their hits and misses on the same display system.

System Functional Specification

The system must carry out the same operations as the classic battleship game, in which two users take turns firing missiles until a player loses all their ships. The system controls when the LED array changes, so that the user knows when it's their turn to fire missiles, and also shows where they have fired and where their ships are.

Operating Specifications

Firstly, the system will allow the user to set up their ships as they want. Then the users will decide together who will go first. Once the decision is made, the user enters whether they are firing missiles or waiting their turn. Based on the user input, the console would change, and the LED matrix will be updated to help guide the user in decision making.

The nRF24L01 will operate at an ultra low power 2Mbps transfer rate, at 2.4GHz ISM (Industrial, Scientific and Medical) bandwidth.

Reliability and Safety Specification

The system is expected to not fail, and has edge cases taken care of, so that regardless of user input, the system will not fail. There is no calculated mean time between failures, as we have yet to encounter any failures. User must avoid touching the circuit system on the DE1-SoC board to prevent any failures of the system, as data transfer within the system requires working circuitry, and wires connections should not be modified.

Our Battleship system meets all the safety specifications required for safe operation for users of all sizes and ages. Switches on the DE1-SoC must be handled with care, and contact with sharp edges on the board should be avoided.

Battleship Requirements Documentation

Abstract

Battleships is a two player game in which the aim is to sink all of the opponent's ships. Players take turns giving coordinates where they wish to fire missiles and take down different sized ships. The first player to sink all the opponent's ships wins.

Introduction

Our system functions as a two player battleships game, in which we prompt user to set up their ships, and communicate coordinates and other information through WiFi. We will rely on the user to communicate through words to decide who will go first, once the ships are set up. The user will use the program console to send coordinates to the other system, and in return the system will receive data from the other system and the console will show whether the user hit or miss the opponent's ships.

Inputs

The system must meet the following input requirements:

- The user must be able to place their ships onto the game board.
    - The user must be able to choose whether they would like to place their ship horizontally or vertically, and pick one coordinate to place their ship accordingly.
        - For example, if the user chooses a vertical ship, they must enter a top coordinate, or a left coordinate for a horizontal ship.
- The user must be able to layout their ships in some acceptable format
    - Acceptable format does not include giving the user just one simple ship layout and forcing them to select it.

We assume that the user will have a keyboard to enter data from.

Outputs

The system must meet the following output requirements:

- Console displaying whether a ship has been hit or missed
- Console displaying whether a user has won or lost
- Any method to show the user their ship layout and where they have shot and missed
    - Preferably using the LED Matrix Array

We assume that the user will have a screen to display console output on.

Major Functions

One of the major functions of this program, aside from actually being able to play Battleships, is the fact you can play Battleships wirelessly. The system includes several functions such as parity checking and memory storage ensuring that data sent and received is accurate and stored for later use. We will be using the SRAM on our NIOS processor to store data, and the nRF24L01 transceiver IC to communicate data between the two systems.

C Code

LED Matrix

```
#define yourboard0 (char*) 0x21090
#define yourboard1 (char*) 0x210f0
#define yourboard2 (char*) 0x210e0
#define yourboard3 (char*) 0x210d0
#define yourboard4 (char*) 0x210c0
#define yourboard5 (char*) 0x21080
#define yourboard6 (char*) 0x210b0
#define yourboard7 (char*) 0x210a0
#define yourshots0 (char*) 0x21070
#define yourshots1 (char*) 0x21060
#define yourshots2 (char*) 0x21050
#define yourshots3 (char*) 0x21040
#define yourshots4 (char*) 0x21030
#define yourshots5 (char*) 0x21020
#define yourshots6 (char*) 0x21010
#define yourshots7 (char*) 0x21000
#define shotsBase (unsigned int) 0
#define boardBase (unsigned int) 16

void ledArrayYourBoard(int i, unsigned char byte) {
    switch (i) {
    case boardBase:
        *yourboard0 = byte;
        break;
    case boardBase+1:
        *yourboard1 = byte;
        break;
    case boardBase+2:
        *yourboard2 = byte;
        break;
    case boardBase+3:
            *yourboard3 = byte;
            break;
        case boardBase+4:
            *yourboard4 = byte;
            break;
        case boardBase+5:
            *yourboard5 = byte;
            break;
        case boardBase+6:
            *yourboard6 = byte;
            break;
        case boardBase+7:
            *yourboard7 = byte;
            break;
        default:
            break;
        }
    }

    void ledArrayYourShots(int i, unsigned char byte) {
        switch (i) {
        case shotsBase:
            *yourshots0 = byte;
            break;
        case shotsBase+1:
            *yourshots1 = byte;
            break;
        case shotsBase+2:
            *yourshots2 = byte;
            break;
        case shotsBase+3:
            *yourshots3 = byte;
            break;
        case shotsBase+4:
            *yourshots4 = byte;
            break;
        case shotsBase+5:
            *yourshots5 = byte;
            break;
        case shotsBase+6:
            *yourshots6 = byte;
            break;
        case shotsBase+7:
            *yourshots7 = byte;
            break;
        default:
            break;
        }
```

```
}
```

Verilog Modules

SRAM

```
//In & out ports consist of 8 parallel wires, an input address of 12 bits chooses the
address to access in the RAM,
//and the system will either read from RAM or write to RAM based on the configuration
of the
//active low input signals, "not output enable" and "read not write"
//This specific RAM consists of 2048 addresses, each containing one byte of data.
module sram (data, address, notOutEn, readnWrite, chipSelect);
	inout [7:0] data;
	input [10:0] address;
	input notOutEn, readnWrite, chipSelect;
	reg [7:0] ram [2047:0];
	 // Read
	assign data = (~notOutEn & ~chipSelect) ? ram[address] : 8'bz;
	//Perform the write operation when read signal is strobed high
	always @(posedge readnWrite)
	//Overwrites the memory at proper address using the input signal on the data bus
	if (~chipSelect)
		ram[address] <= data;
endmodule
```

LED Matrix

```
 /* Adapted from the LED matrix tutorial available
https://courses.cs.washington.edu/courses/cse369/15au/labs/LED_Array_Tutorial.pdf */
 module ledDriver (greenDriver, redDriver, rowSink, redArray, greenArray, boardKey,
clk);
    input boardKey, clk;
    input [7:0][7:0] redArray, greenArray;
    //output reg [7:0] led; // TODO debugging
    output reg [7:0] redDriver, greenDriver, rowSink;
    reg [2:0] count;
    reg [19:0] slowClk;
    always @(posedge clk)
    begin
        slowClk = slowClk + 1;
    end
    always @(posedge slowClk[14])
    begin
        count <= count + 3'b001;
    end
    always @(*)
    begin
```

```
        case (count[2:0])
                     3'b000: rowSink = 8'b11111110;
                     3'b001: rowSink = 8'b11111101;
                     3'b010: rowSink = 8'b11111011;
                     3'b011: rowSink = 8'b11110111;
                     3'b100: rowSink = 8'b11101111;
                     3'b101: rowSink = 8'b11011111;
                     3'b110: rowSink = 8'b10111111;
                     3'b111: rowSink = 8'b01111111;
        endcase
        redDriver = (~boardKey) ? redArray[count] : 8'b0;
        greenDriver = (boardKey) ? greenArray[count] : 8'b0;
    end
 endmodule
```

Serial Communications

```
module startBitDetect(recv_en, data_in, char_received);
    output reg recv_en = 1'b0;
    input data_in, char_received;
    always @(negedge data_in or posedge char_received)
        if (char_received) recv_en <= 1'b0;
        else recv_en <= 1'b1;
endmodule

module bsc (sr_clock, clk16x, bsc_en);
    input clk16x;
    input bsc_en;
    output sr_clock;
    reg [3:0] bsc_count = 4'b0;
    assign sr_clock = ~bsc_count[3] & bsc_en;
    always@(posedge clk16x)
        if (bsc_en) bsc_count <= bsc_count + 1'b1;
        else bsc_count <= 4'b0;
endmodule

/* Receiving 10 bits */
module bicReceive (seq_complete, sr_clock, bic_en, char_read);
    input bic_en, sr_clock, char_read;
    output seq_complete;
    reg [3:0] bic_count = 4'b0;
    always@(posedge sr_clock or posedge char_read)
        if (char_read) bic_count <= 4'b0;
        else if (bic_en) bic_count <= bic_count + 1'b1;
        else bic_count <= 4'b0;
    assign seq_complete = bic_count[3] && ~bic_count[2] && bic_count[1] &&
~bic_count[0];
endmodule
```

```
/* Transmitting 10 bits */
module bicTransmit (seq_complete, sr_clock, bic_en);
    input bic_en, sr_clock;
    output seq_complete;
    reg [3:0] bic_count = 4'b0;
    always@(posedge sr_clock or negedge bic_en)
        if (~bic_en) bic_count <= 4'b0;
        else if (bic_en) bic_count <= bic_count + 1'b1;
        else bic_count <= 4'b0;
    assign seq_complete = bic_count[3] && ~bic_count[2] && bic_count[1] &&
bic_count[0];
endmodule

/* 16x clock for counting middle of each bit */
module clock16x(CLOCK_50, clk);
       input CLOCK_50;
       output reg clk = 0;
       reg [7:0] count = 8'b0;
       always @(posedge CLOCK_50) begin
              if (count == 8'b10100010) begin
                          count <= 8'b0;
                          clk <= ~clk;
              end
              else begin
                          count <= count + 1'b1;
                     clk <= clk;
              end
       end
endmodule

/* Parallel to serial shift register */
module PISOSR10bit (bit_out, load, sr_clock, data, rst);
    input sr_clock, load, rst;
    input [7:0] data;
    output bit_out;

    reg [9:0] buffer = 10'b1111111111;

    assign bit_out = buffer[9];

    always @(posedge sr_clock or posedge rst)
        if (rst) buffer <= 10'b0;
        else if (load) buffer <= {1'b0, data, 1'b1};
        else buffer <= {buffer[8:0], 1'b1}; // shift out

endmodule

/* Serial to parallel shift register */
module SIPOSR10bit (dataout, datain, sr_clock, rst);
```

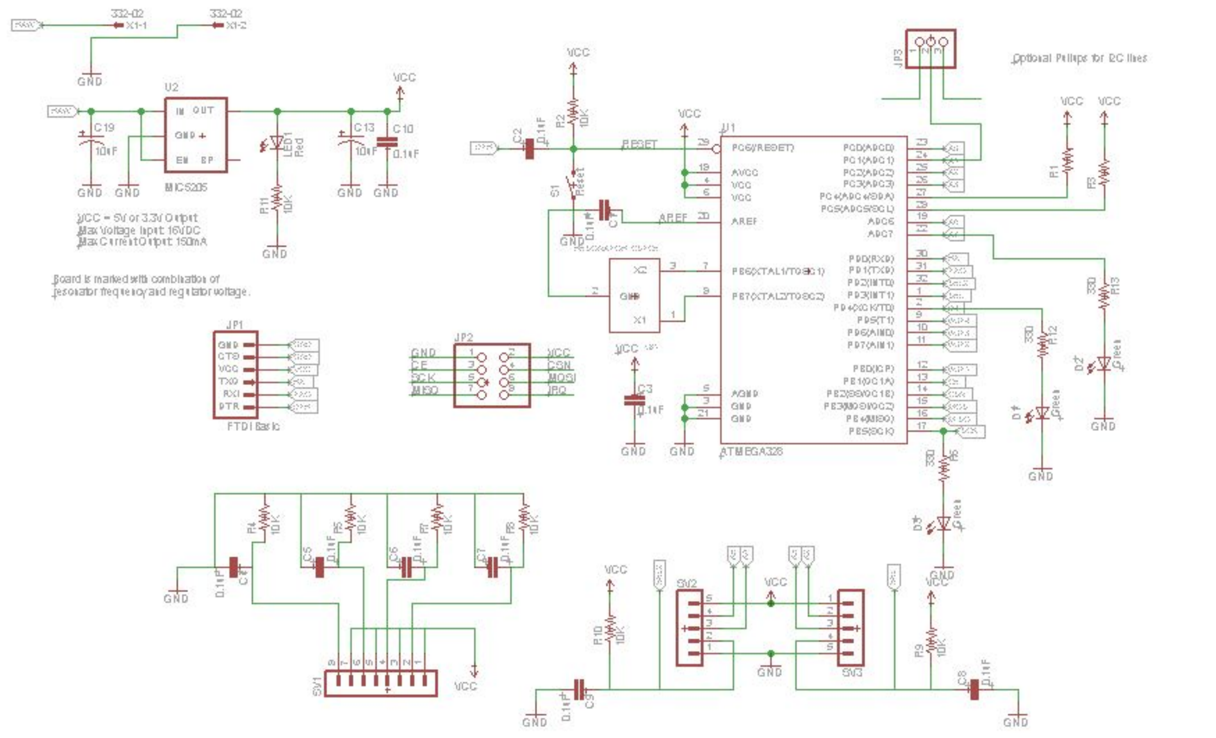
```
    output [7:0] dataout;
    input datain, sr_clock, rst;

    reg [9:0] buffer = 10'b0;

    assign dataout = buffer[8:1];

    always @(posedge sr_clock or posedge rst)
        if (rst) buffer <= 10'b0;
        else buffer <= {buffer[8:0], datain};
endmodule
```

Wireless Communication

The following is the Arduino code for the

```
// Including necessary libraries
#include <SPI.h>
#include <Mirf.h>
#include <nRF24L01.h>
#include <MirfHardwareSpiDriver.h>

uint8_t data[3];
// Global Board Select Variable.
// One board must be 1, other must not be 1
int board = 0;
char dump;

void setup () {
   // Set the Baud rate to match DE1
   Serial.begin(9600);
   Mirf.cePin = 9;
   Mirf.csnPin = 10;
   data[2] = '\0';

   // Mirf library commands for initialization
   Mirf.spi = &MirfHardwareSpi;
   Mirf.init();
   if(board == 1) {
      Mirf.setRADDR((byte *)"clie1");
   } else {
      Mirf.setRADDR((byte *)"serv1");
   }

   Mirf.payload = sizeof(data);
   Mirf.config();
   delay(50);
   // Clear Arduino input serial buffer
   clearSerial();
}

void loop() {
   // Wait for either serial data or wireless data to enter the State machine
   if(Serial.available() > 0) {
      sendData();
   } else if(Mirf.dataReady()) {
      receiveData();
   }
}

void sendData() {
   // Status LED to show state
   digitalWrite( 4 , HIGH);
      if(board == 1) {
      Mirf.setTADDR((byte *)"serv1");
   } else {
      Mirf.setTADDR((byte *)"clie1");
   }

   // Wait for three bytes of serial data and read them into the data array
   while(Serial.available() <= 0);
   data[0] = Serial.read();
   while(Serial.available() <= 0);
   data[1] = Serial.read();
   while(Serial.available() <= 0);
   data[2] = Serial.read();

   // Clear any extra inputs from input buffer
   clearSerial();
   Mirf.send(data);
   while(Mirf.isSending()) {
      digitalWrite(4 , HIGH);
   }
   delay(100);
   digitalWrite(4 , LOW);
   // Go to next state
   receiveHitMiss();
}

void receiveData() {
   digitalWrite(4 , LOW);
   while(!Mirf.dataReady());
   Mirf.getData(data);
   delay(50);
   Serial.write(data[0]);
   // Delay for suspense to see whether the coordinate was a hit or not
   delay(500);
   Serial.write(data[1]);
   delay(50);
   Serial.write(data[2]);
   delay(50);
   // Go to next state
   sendHitMiss();
}


void sendHitMiss() {
   digitalWrite(4 , HIGH);
   if(board == 1) {
      Mirf.setTADDR((byte *)"serv1");
   } else {
      Mirf.setTADDR((byte *)"clie1");
   }

   while(Serial.available() <= 0);
   data[0] = Serial.read();
   // Set Null terminator
   data[1] = '\0';

   // Clear Arduino input serial buffer
   clearSerial();

   Mirf.send(data);
   while(Mirf.isSending()) {
      digitalWrite(4 , HIGH);
   }
   digitalWrite( 4 , LOW);

   // Go to next state
   sendData();
}

void receiveHitMiss() {
   digitalWrite(4 , LOW);
   // Wait for wireless data
   while(!Mirf.dataReady());
   Mirf.getData(data);
   delay(50);
   Serial.write(data[0]);
   delay(500);
   Serial.write(data[1]);
   delay(50);
   // Go to next state
   receiveData();
```

```
}

// Clears the Arduino serial input buffer to avoid storing
// previous or extra serial data
void clearSerial() {
   delay(1000);
   while(Serial.available() > 0) {
      dump = Serial.read();
      delay(50);
   }
}
```



U2
MIC5205
VCC = 5V or 3.3V Output
Max Voltage Input: 16VDC
Max Current Output: 150mA
Board is marked with combination of resonator frequency and regulator voltage.
JP1
FTDI Basic
JP2
U1
ATMEGA328
RESET
AREF
Optional Pullups for I2C lines
JP3
SV1
SV2
SV3
GND
VCC

Figure 10: Eagle CAD Schematic for Wireless PCB

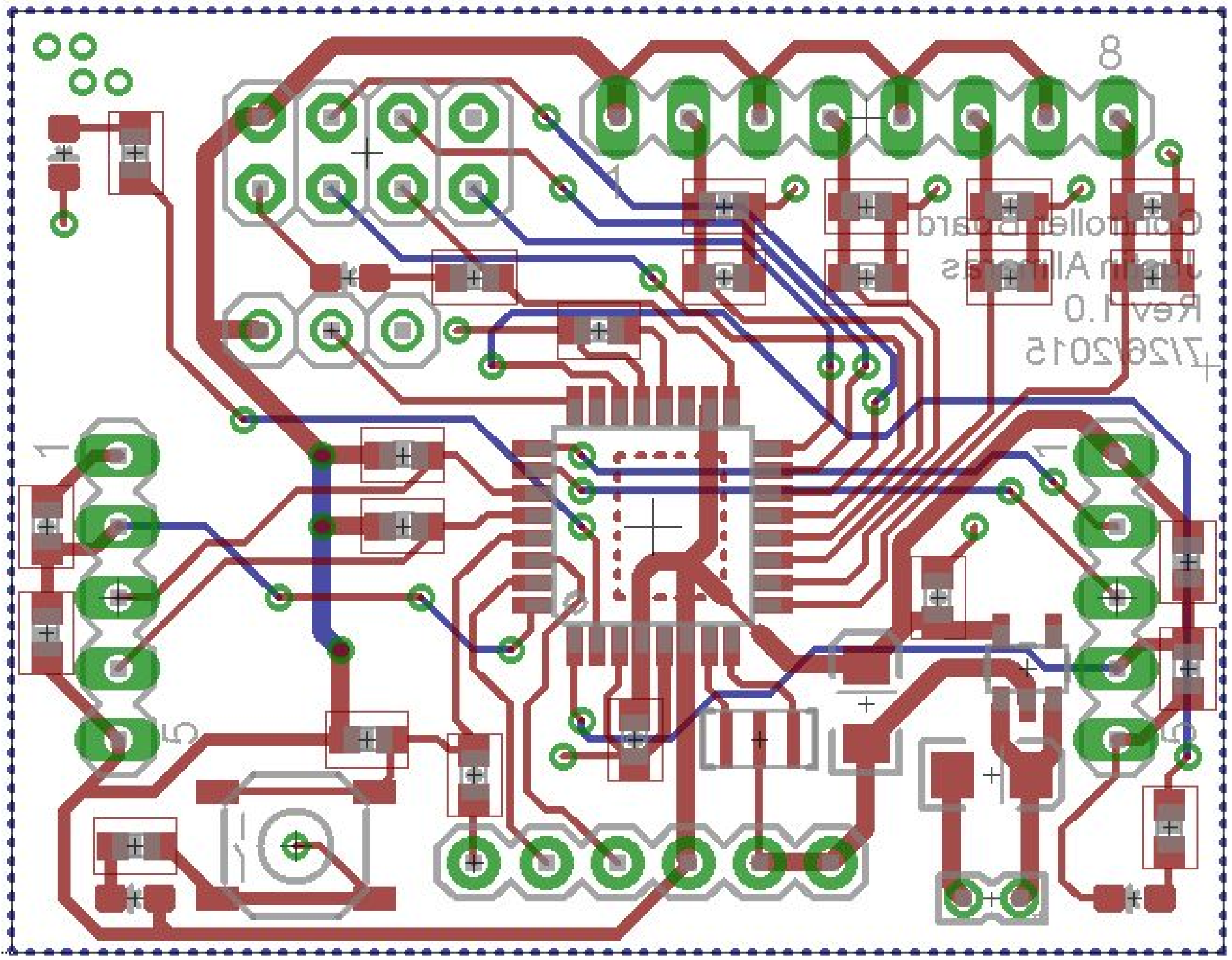


Figure 11: Eagle CAD Layout for Wireless PCB

Who Did What

Sean:

- Worked on C code (parity generation/checking, send/transmit strings and characters)
- Implemented and tested LED matrix driver assignment in C and Verilog
- Helped debug Verilog SRAM design
- Contributed to lab report

Justin:

- Wireless design and testing
- Verilog design for UART
- LED Matrix testing
- Power switch implementation
- Contributed to lab report

Vishesh:

- Wrote Design Specification and System Requirements
- Helped debug parity bit checking and transmission in lab 4
- Contributed to C program for Battleship
- Worked on wireless data transmission
- Contributed to lab report