ECE 540 Final Project
I've Got Heart:
A Wireless Heart Rate Monitor

Ryan Bentz
Ryan Bornhorst
Andrew Capatina
Alex Olson

2019-March-21

Table of Contents:

1.0 Introduction

The I've Got Heart Project is an attempt to build an SoC-based wireless heart rate monitor. **Figure 1** shows the high level diagram of the design. An analog heart rate sensor sends data to an ADC which is converts the signal into digital values. A 2-wire interface implemented in the FPGA extracts the digital values in a SPI-like fashion. The interface is part of a peripheral on the AHB-Lite bus of the MIPSfpga softcore processor in the FPGA. The module extracts the data values and packs them together for the application CPU to read. The application CPU uses the values to calculate the real-time BPM, max BPM, min BPM, and detect heart beats in real-time. Calculated heart rate statistics are written to a second peripheral which manages a 2-wire SPI-like interface that transmits the data to the ESP8266 which is then propagated to Google Firebase. Not only will the CPU write data to a wifi peripheral, it will also be responsible for monitoring the push button input the user can provide to cycle through the different kinds of statistics that may be displayed on the seven segment display; our implementation displays minimum, maximum, and average beats per minute.

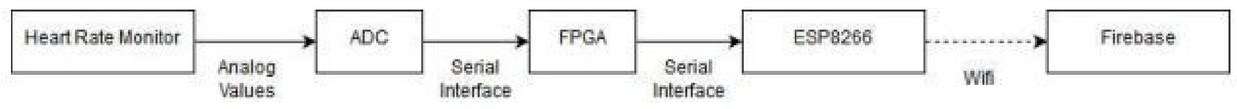

Figure 1: High level schematic of heart rate monitor.

2.0 Hardware Design

The HDL implements two SPI-like interfaces to interact with ADC for the heart rate monitor and the ESP8266. Both interfaces are connected to the MIPSfpga softcore processor via AHB-Lite Peripherals. Figure 2 shows the new memory map for the AHB-Lite Peripherals. Three new peripherals have been added: Heartbeat, SPI_Master, and Timer. The Heartbeat manages the interface between the peripheral gathering data from the ADC. The SPI_Master handles transmitting data to the ESP8266, and Timer creates a timer-counter that is used to detect when it is time to calculate the BPM.

Register Name	Address	Description
SEVENSEG_ADDR	0x1F70 0000	Controls Seven Segment Displays
LED_ADDR	0x1F80 0000	Controls Switch LEDs
HEARTBEAT_READ	0x1F90 0000	Read the packed data values from ADC
HEARTBEAT_RDY	0x1F90 0004	Read signal from peripheral that data is ready
HEARTBEAT_ACK	0x1F90 0008	Write signal to peripheral that data was read
SPI_MASTER (ESP8266)	0x1F9A 0000	Write 2 bytes to peripheral for sending to ESP8266
TIMER_ACK	0x1FA0 0000	Write to acknowledge timer overflow
TIMER_RDY	0x1FA0 0004	Signal from timer that overflow occurred

Figure 2: AHB-Lite Peripheral Addresses

2.1 Physical Design

2.2 Heartbeat Sensor

Figure 3 shows the physical connections of the heartbeat sensor, ADC and FPGA.

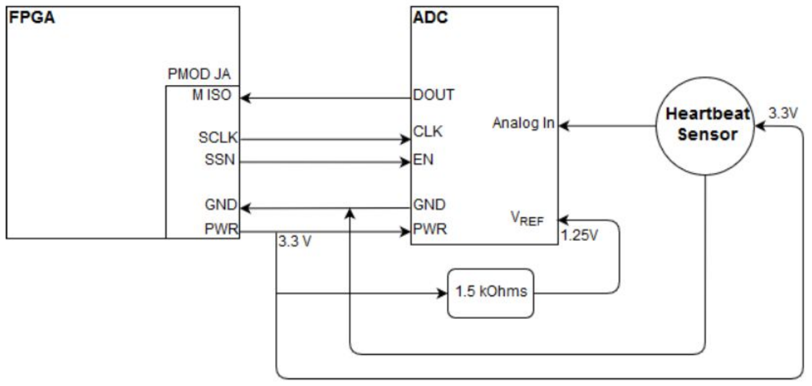

Figure 3: Physical Layout of Heart Rate Monitor and ADC

The heartbeat sensor is a Pulse Sensor Amped device that includes the sensor combined with amplification and noise-cancelling circuitry. The output voltage ranges from 20mV - 30mV.

Figure 4 shows the circuitry for the sensor.

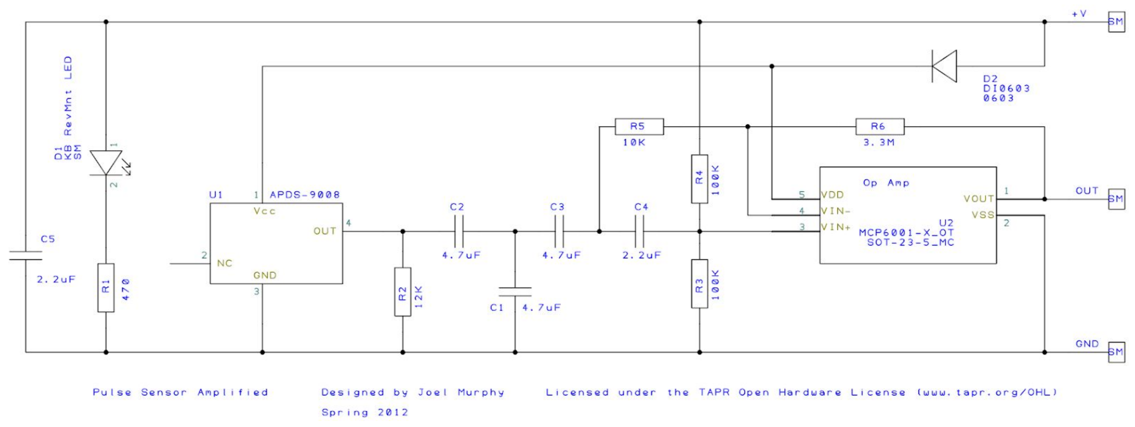

Figure 4: Circuitry for Heart Rate Sensor

The ADC is a 10-bit ADC from Microchip in a dual-inline package. The specs are as follows:

Model: MCP3001

- Serial Data: 10-bit numbers that take 13 clock cycles to acquire
- Numbers will be truncated to 8-bits
- A clock rate of 1.05MHz can be used with 3.3V supplied from FPGA
- V_{REF} = 1024 * LSB size (formula from data sheet)
- If LSB [bit 0] is designed for 1mV, then bit 7 will be 128mV
- V_{REF} = 1024 * 1mV = **1.024V**

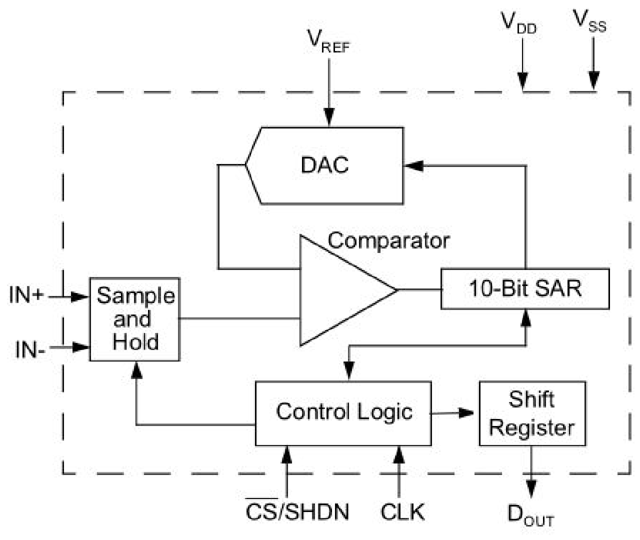

Figure 5: ADC Internal Circuitry

2.3 FPGA-ADC Interface

The purpose of this peripheral was to buffer the digital output into a location that could be transferred to the AHB-lite for the mipsFPGA to read. Interacting with the ADC first required tuning it to be sensitive to the voltage range output by the heartbeat sensor. The sensor was capable outputting a negative voltage, but this was able to be ignored by setting the V- input to ground. As it was shown in the calculations, setting the reference voltage to 1.024 volts gives a sensitivity range of 0 - 128mV. In order to achieve this voltage drop, an output current of 1.365mA was recorded and used to determine that a resistor with 1.5kΩ of resistance could provide the needed voltage drop. The ADC also requires that its slowest conversion rate (1.05MHz) needs a minimum voltage of 2.7 volts. The FPGA outputs about 3.3 volts so that is more than sufficient to meet the requirements of the ADC.

2.3.1 ADC Sampling Rates

The ADC outputs digital data utilizing the SPI protocol. As specified by the ADC datasheet, when given an active low enable signal, the chip will begin the ADC process on the next leading clock edge. The first two clock edges of the transfer are used to convert the analog signal to a 10-bit digital value using Successive Approximation Register. The 3rd clock edge outputs a null bit. The next 10 clock edges output the 10-bit number with the MSB being shifted out first. After the 13 clock cycles, the active low enable signal is asserted high and the process is terminated. The FPGA controls the SPI handshaking as well as determining the sampling rate. In order to help synchronize the ADC clock with the sampling rate, a 16-bit counter was used which was drive by an 8.4MHz clock from the clock wizard. Bit 3 of the counter provides a 1.05 MHz clock for the ADC and bit 15 was used to enable the sampling at 128 Hz which is then ended once the counter has had bit 3 toggle another 13 times after bit 15 is set (The process starts when the counter is 16’h8000 and ends sampling at 16’h8068). This provided sufficient resolution for the peak detection algorithms in the software.

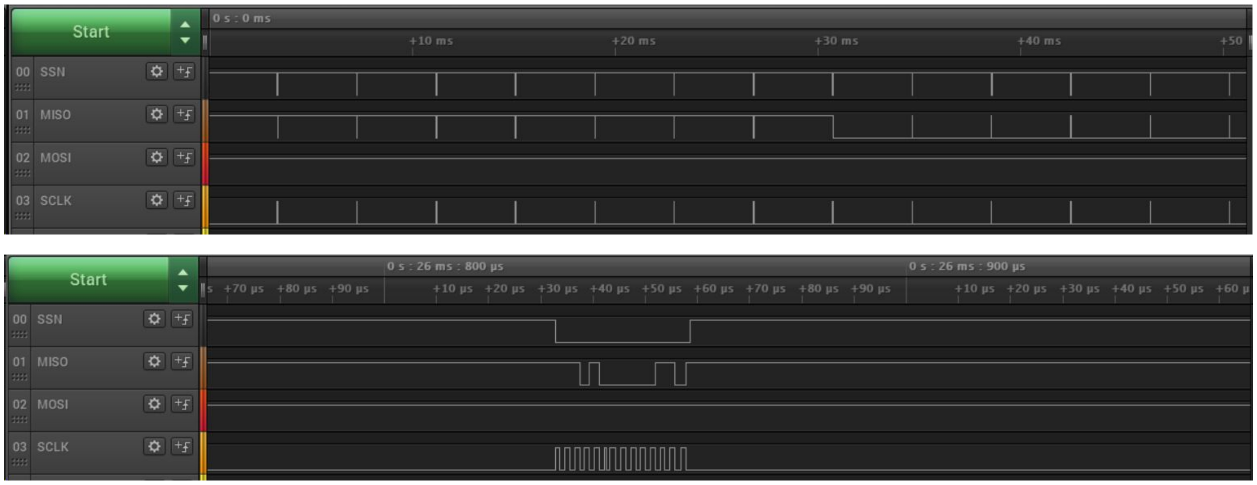

Figure 6: Logic Analyzer Output of Heart Peripheral and ADC Data

2.3.2 Heartbeat Interface with AHB

Once the sampling rates were resolved, that raw data actually has to be sent to the mipsFPGA. To help ease the software programming, the hardware reversed the MSB first data into little endian format and truncated the 10-bit numbers down to 8-bit numbers. Since the AHB uses 32-bit words, a buffer was created that holds four 8-bit numbers. A handshaking protocol was initiated with the mipsFPGA once this buffer was filled. Since the data is being acquired relatively slowly (one 32-bit word is filled about every 31.25 milliseconds) the protocol could be initiated safely every time the buffer filled with four converted numbers. A fully-interlocked asynchronous handshaking protocol was used to satisfy these requirements. When the buffer is ready, a READ_RDY signal is sent to the processor using an address on the AHB. The software polls this signal each time through the loop and asserts a READ_ACK signal to the heartbeat module, letting it know that the data is ready to be received. Upon receiving the READ_ACK signal, the heartbeat module then de-asserts the READ_RDY signal and outputs the 32-bit buffer data onto the IO_HEARBEAT address to be read by the mipsFPGA. Once the processor reads the data, it de-asserts the READ_ACK signal and waits for the next four numbers to come at the following handshake.

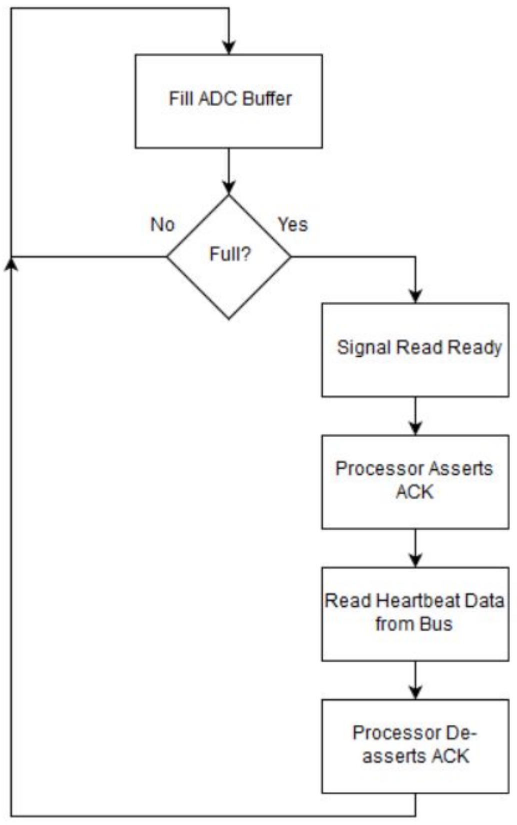

Figure 7: Handshake Protocol between Heart Peripheral and Application Program

2.4 FPGA-ESP8266 Interface

This interface serves as the bridge between the application program and the ESP8266. The interface is implemented as an AHB-Lite bus peripheral that provides a 16-bit register for the application to write data to. The peripheral takes the data and sends it out on the 2-wire interface to the ESP8266. Data transmission begins when the register is written to the register. The peripheral the write enable signal goes high along with the start transaction signal. While the start signal is high, heart rate data along with a synchronized clock signal gets driven out serially to the ESP8266 device through the IO ports on the Nexys board. Two bytes of data get sent out to the wireless device, the first of which is used to inform the ESP8266 software of the type of heart rate data coming in, and the second which is the actual heart rate data. After a transaction has been made, the peripheral asserts the stop transaction signal high to allow another transaction to take place.

2.5 Timer

A timer module was implemented on the AHB-Lite bus. The timer is used for the application program is located in timer_count.v. The counter variable used has a limit of 6 seconds. Once 6 seconds elapses a value is sent to the CPU that can be read. The CPU may send a value to this module for resetting the counter. The file for sending data between cpu and hardware is mfp_ahb_counter.v.

3.0 Software Design

3.1 MIPS Application Program

The high level flow of the application program is shown in **Figure 8**. The application program persistently checks if either new data is ready from the ADC or if the 6 second timer has overflowed.

If the timer has overflowed, the program will take the number of beats counted since the last timer overflow and calculate the Beats Per Minute. A typical heart rate is going to be in the 60-100 BPM which equates to just over 1 beat per second. The 6 second interval was chosen because it allows enough beats to be counted to gauge an accurate enough BPM but also allows easy conversion to BPM by simply multiplying by 6. The new BPM is compared to see if a new min or max was reached and data is written to the seven segment display. This would also have been where data would be transmitted to the ESP8266 is the serial connection was working.

If new data is ready from the ADC, the program will read the data and let the peripheral know the data was received. It will then pass the data through the beat detection algorithm (see

Section 3.1.1) and then the program returns to checking for new data or waiting for the timer to overflow.

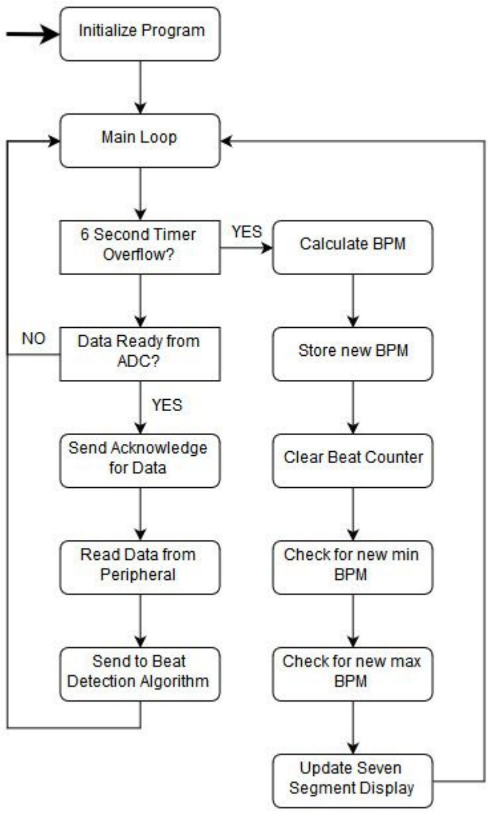

Figure 8: High level flow diagram for software

3.1.1 Beat Detection Algorithm

The Beat Detection Algorithm analyzes the digital values received from the ADC and determines if a beat has been detected. The typical waveform for a heartbeat contains a peak and a trough. However, due to the design of the heart rate monitor-to-ADC connection, the negative voltages are converted to zeros and only peaks are available digitally. A beat is defined as the transition from successively increasing values to decreasing values. As the digital values increase on the rising edge of the peak, a transition is reached where the values decline all the way to the trough. The transition in values indicates a peak was reached and a beat is detected.

Data is available from the ADC peripheral as 32-bit words with 4 readings packed in each word. The algorithm splits each word into the 4 distinct readings and compares them with the "last highest reading received". If the read value is greater than the "highest value" the highest value is overwritten with the new highest value.

If the read value is lower than the "highest value", a transition is identified and beat detected. When a beat is detected, a counter is incremented that keeps track of the number of beats and a flag is set. The Beat Flag member keeps track of the state of the readings (if they are increasing or decreasing). This keeps each successive reading on the downslope of the ADC signal from triggering the "read val < highest val" conditional and incrementing the beat counter. On the downslope of the ADC readings, the stored "highest value" is also written with the new low value. When the low-to-high transition is reached, the Beat Flag is cleared in preparation for a new high-to-low transition.

Figure 9 shows the algorithm for the beat detection algorithm. When, the application program senses enough time has passed, it collects the number of beats that were detected to calculate the BPM. The beat counter is reset each time the application reads the number of beats and process repeats.

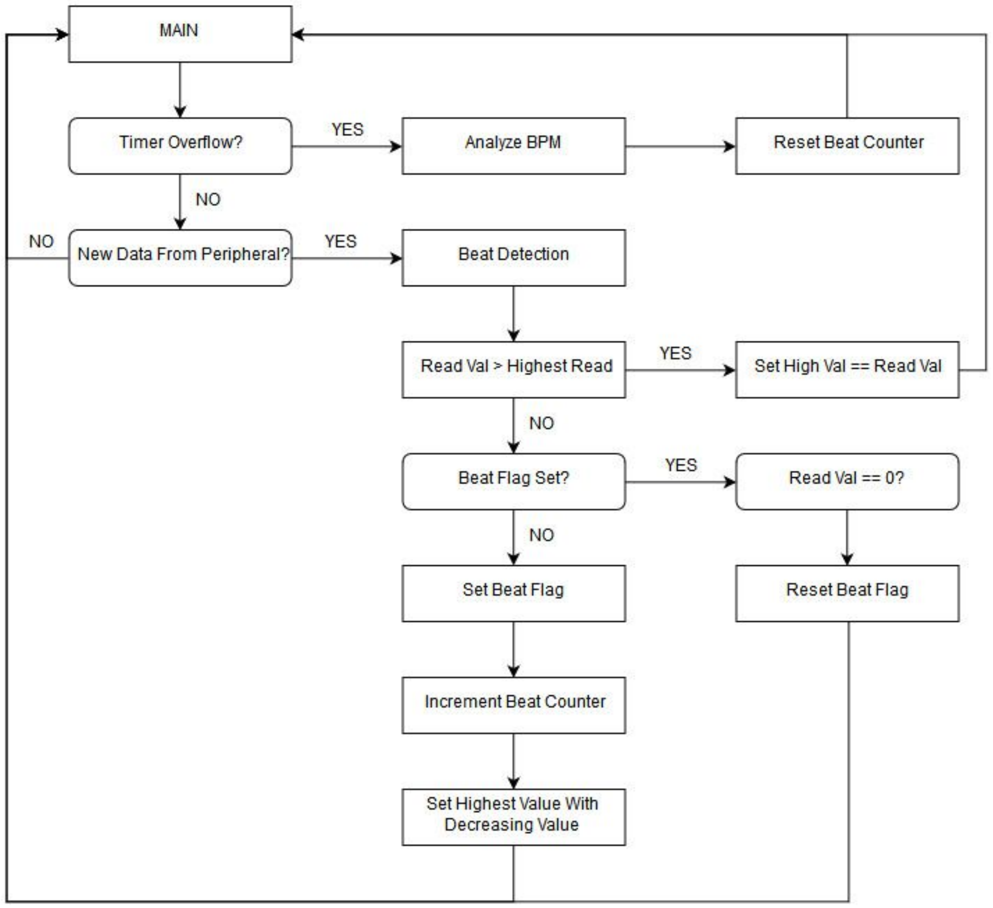

Figure 9: Beat detection algorithm

3.2 ESP8266 Interface Application

There are many ways to implement code on the ESP8266 but our device uses the Arduino software framework in order take advantage of existing libraries to facilitate communication with Firebase. The program initializes two pins for the 2-wire serial interface: CLOCK and DATA. The FPGA has master control of the serial communication and the ESP8266 acts as the listener. The ESP8266 sets an interrupt on the clock line with a rising edge trigger. When the FPGA drives the clock signal high, the ESP8266 program reads the state of the data line and appends it to the LSB of a 16-bit word. The bits in the 16-bit word are shifted on every read until 16 consecutive reads have occured. When the 16-bit word is received, the program splits the word into two bytes. The second byte is the Data Byte and the first byte is a Tag Byte which indicates the data member in Firebase that the data is for. The application then uploads the data to the Firebase data member. After the data is transmitted, the program resets its counters and waits for the next transmission.

3.3 Android Application

The Android application is a single activity application that displays heart rate data and flashes a heart symbol when a beat is detected. Data and beat detection information is read from a Firebase database that is written to by the ESP8266. The app sets event listeners for the members in the Firebase database and when a change is detected in the database, the app reads the new values and updates the TextViews showing the values.

The flashing of the heart symbol uses an Animation Drawable to perform the animation. When a beat is detected, the animation drawable process is started. The Animation Drawable works by cycling through a series of images displayed in an ImageView. Each image in the animation can be given an user-defined length of time to be shown. For the heart beat animation, only two images are used: a red heart and a light-pink heart. The red heart is shown for a 400ms and then the light-pink heart for 10ms. The animation ends on the light-pink heart and waits for the next start of the animation.

4.0 Results

4.1 Proposed Results

The final project proposal set the goal of a successful demonstration as being able to show the heart rate calculation on the seven segment display and on the Android application.

4.2 Actual Results

For the demo, we showed the heart monitor updating its calculations on the seven segment display only. At that time, we were not sending any information from FPGA to the Android app. However, we did demonstrate the Android app working with simulated heart rate values being generated by the ESP8266.

5.0 Conclusion

There were quite a few challenges we had to overcome for this project. All of the problems were solved except for one. We couldn't resolve the ability to send information from FPGA to ESP8266. This was a frustrating problem because we had simulated the HDL to show the correct waveform was being generated. We, then synthesized a dummy counter in the top-level module to prove that signals could be sent out on the PMOD pins. However, attempting to generate the signals from deep in the hierarchy from the SPI peripheral module yielded no results. The current working theory is that there is a problem when trying to generate signals from a module deep in the MIPSfpga hierarchy and/or there is a problem with using a 50 MHz clock source for the 2-wire serial clock. We think that one of the reasons might be that the logic analyzer was too slow to sample the data, however, we would have still seen some kind of signal changes for that to be the case.

Andrew Capatina: At first I was able to make a lot of progress in developing the high level framework of the MIPS core program. I made the framework needed to cycle between the different kinds of data being displayed when given realistic heart rate values. However progress began to stall when I needed to develop the beat detection algorithm. Essentially, I didn't know what kind of data I was expecting to receive; therefore I ended up putting off this portion until closer to demo day. What I learned here was I need to ask my teammates more questions in the future so I can continue making progress. I believe if I mentioned my struggles to my team members earlier they would've been able to provide me the information I needed.

Ryan Bentz: This project sounded a lot better in my head or when just talking about it. The issue of the 2-wire interface not sending out any signals to the ESP8266 was incredibly frustrating because it went against everything we knew and learned in the class. The overall project blew up into much more work then we were expecting and I think is another example of how hard it is to estimate the time required for some these projects. Ultimately, our downfall was Git management. We were within striking distance of our goal but file merging issues at the 11th hour really took the wind out of our sails and became the driving reason behind us not meeting the project goal.

Alex Olson: I had an absolute blast doing this project probably because I was lucky enough to have the fun part of hardware design. There were definitely some frustrations and uncertainty. Namely, I was concerned about the actual data that I was buffering and how I could package and send it in a usable form to the software. Leaving the ADC on full blast would leave us with a million samples to process every second which was way more than necessary for a pulse. My biggest concern was how small I could make my sample sizes combined with how often I could send data to the processor at a rate that would not overload it with extra, unnecessary analysis. It turned out to be a really fun challenge to couple this problem along with ADC resolution as well as the SPI interface to the FPGA.

Ryan Bornhorst: I enjoyed working on this project even though the portion of the project that I was in charge of ended up not working as expected. I enjoy working with FPGAs and SystemVerilog/Verilog but am not very fond of the MIPSfpga related work. I think that I will have a blast working on the projects in ECE-544 because I really enjoy embedded programming in C. Even though the interface between the FPGA and ESP8266 did not work out the way we expected, I think we were very close to getting this portion of the project working. We were able to get simulation to work as expected but could not properly propagate the signal through the IO ports on the top module. Even though it didn’t work out the way that we wanted to, I think all of us learned a lot while working through this project and if we were to continue working on it, we could all have it up and running within a day or two.