E155 Project Final Report

Andrew M. Pham, Dana ShangGuan

December 13, 2019

Abstract:

Using accelerometers on large structures, such as dams, can be cumbersome because of the hundreds of feet of cabling involved. Wireless accelerometers avoid the issues associated with cable management. Thus, a wireless accelerometer setup is investigated in this report. This setup consists of an analog accelerometer, and a Bluetooth module. The user is able to interface with this setup using MATLAB, and can collect filtered data from the accelerometer through Bluetooth in real-time. By executing code through MATLAB, the system begins wireless data collection. First, the analog accelerometer is sampled through the microcontroller, and this data is sent to the FPGA to be filtered. Next, after the FPGA is finished filtering, this output is transmitted back to the microcontroller in order to be sent via Bluetooth module. The Bluetooth module transmits the filtered data to MATLAB, where it is stored and is ready for data processing.

Introduction

We are working in a civil engineering research group and often use accelerometers in our work. A common problem with these accelerometers is that they require a significant amount of cabling, which can be cumbersome. This project will serve as an exploration into moving into wireless accelerometer sensing capabilities. Therefore, the purpose of this project is to create a wireless accelerometer prototype with band-stop filtering capabilities. We will be measuring acceleration data from one axis in the accelerometer (i.e., x, y, or z-axis). We will use hardware acceleration to filter out 60 Hz signals in real-time, and send it over to a computer to be stored in a data file using Matlab Bluetooth toolbox. The microcontroller should be able to sample the accelerometer at a sampling frequency of more than 400 Hz, to avoid aliasing for signals under 200 Hz.

The wireless accelerometer system will use both the ATSAM4S4B microprocessor onboard, as well as the Altera Cyclone IV FPGA. In addition, new hardware not used in E155 will be used: the BlueSMiRF Silver Bluetooth Module, as well as the Adafruit ADXL335 analog accelerometer. Figure 1, as shown below, is a block diagram of the overall system, which provides a general overview of the interfaces between the hardware.

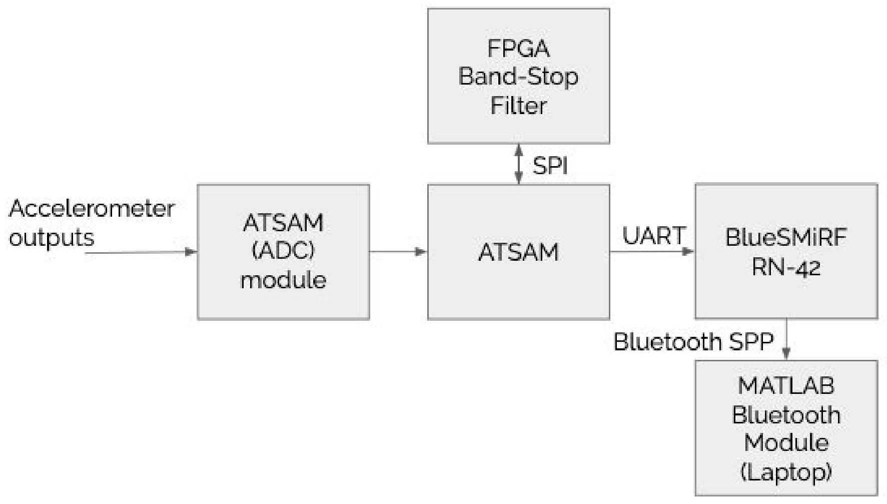


Figure 1. Block Diagram of Overall System and Interfaces

The ATSAM4S4B functions as the main communications hub between all of the different components on the board. The onboard ADC module is used to sample data directly from the analog accelerometer. The ATSAM then processes this data, then passes it to a finite impulse response filter (FIR) implemented on the FPGA. These data are sent over to the FPGA using SPI communication, and after the filter is done, they are sent back to the ATSAM4S4B. The ATSAM4S4B then communicates with the BlueSMiRF over UART communication, and sends the filtered data over Bluetooth to MATLAB.

New Hardware

BlueSMiRF Silver

The system makes use of the BlueSMiRF Silver module. This module communicates with the microcontroller using UART, and both MATLAB and the ATSAM communicate with the BlueSMiRF at a baud rate of 19,200 bits per second. In order to configure the BlueSMiRF properly, refer to reference [1], which shows the commands that must be sent to the module once a Bluetooth connection has been established. Filtered acceleration data is sent to the Bluetooth Rx pin and the Bluetooth module sends it via serial communication to a computer with Bluetooth capabilities. We take advantage of MATLAB's Bluetooth toolbox in order to communicate with this hardware.

ADXL335

The system uses the ADXL335 triple-axis accelerometer to acquire an analog acceleration data continuously, where the analog voltage output ranges between 0-3.3V. A 0g acceleration will be represented as 1.65V, and its acceleration sensitivities are 0.3 V/g. There are three output pins from the accelerometer, labeled as *Xout*, *Yout*, and *Zout*, respectively. Refer to reference [2] for further details on the module.

Finite Impulse Response Filter Design

The band-stop filter used in this system will be a finite impulse response (FIR) filter. For information about the implementation of digital filters in hardware, refer to reference [3]. FIR filters are casual, so they rely only on the present and past inputs, which enable real-time filtering of a device. A band-stop filter that filters signals at 60 Hz was designed in Matlab, and its magnitude plot is shown below. As shown in figure 2 below, the filter is capable of attenuating electrical noise at 60 Hz by -40.4 dB. It has 153 coefficients, and its impulse response has a length of 153.

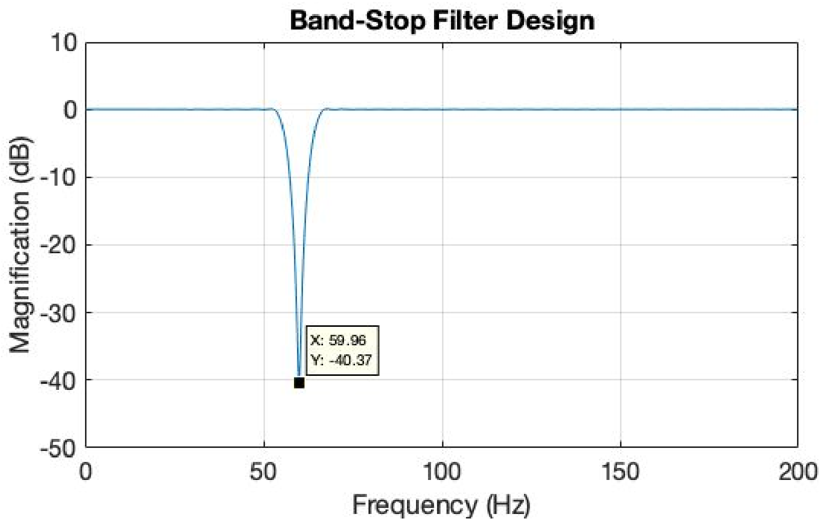


Figure 2. Magnitude Plot of the Band-stop Filter

Schematics

Circuit Schematic

Figure 3 below is the hardware schematic for the wireless accelerometer system. The ADXL335 triple-axis accelerometer is powered by the 3.3V output from the uMudd board, and it is continuously outputting analog data to pin AD0-AD2, which will later be sampled and converted to digital data by the ADC module. The data will be sent over to FPGA through internally connected pins, and the output data will be sent over using the RN42 Bluetooth module. The RN42 BlueSMiRF Silver Bluetooth is connected to the uMudd Board through its Tx Rx pins to the corresponding Rx Tx pins in the SAM4S4B. It will receive the processed data in the Rx pin, and will send it wirelessly to a laptop via serial communication.

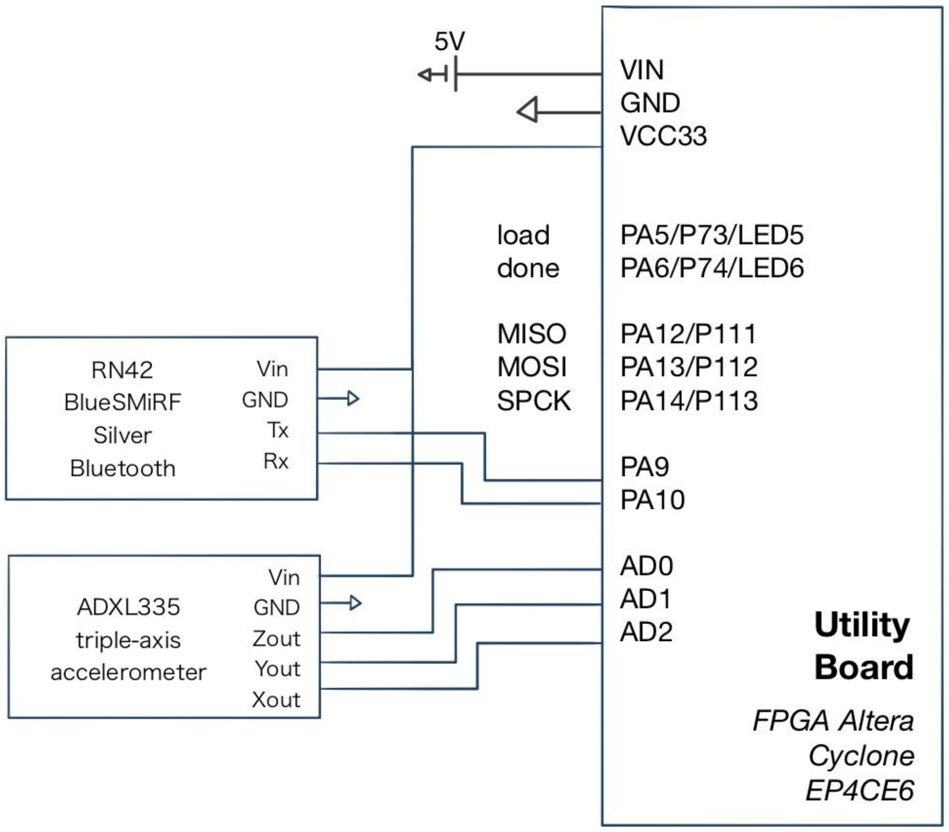


Figure 3. Hardware Schematic

Top-Level Routine

The system can be broken down into three subcomponents. They are the accelerometer module, the Bluetooth module, and the band-stop filter that can filter out 60 Hz signals. Figure 4, shown below, is a high-level overview of our system.

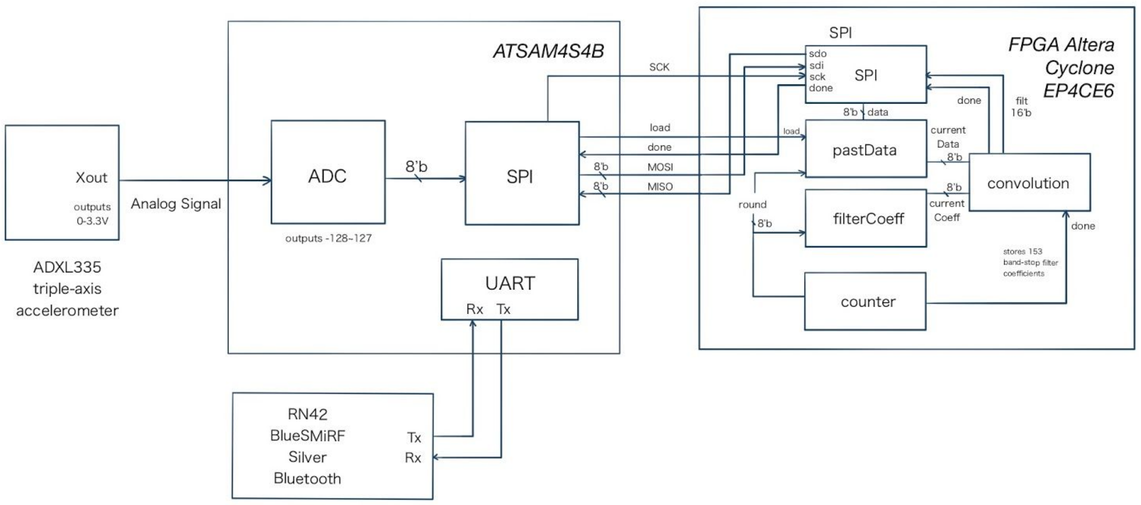


Figure 4. Top-Level Modules Block Diagram

Once the ATSAM is booted, it starts by sampling the accelerometer's outputs from one axis. These outputs are voltage values in floating-point notation, which are quantized into signed chars for filtering in the FPGA. The signed chars are sent over SPI to the FPGA, which filters the data in real-time. Once the filtering is complete, the data are sent back as shorts to the microcontroller, which are then transmitted over to the Bluetooth module over UART for wireless transmission to a computer. The conversion of these shorts into units of acceleration is left to the computer to handle after transmission. The document top.c, attached in the appendix, provides a detailed overview of the routine that the microcontroller uses to control the system.

Microcontroller Design

The ATSAM4S4B serves as the primary interface between all of the hardware in this system. It is therefore in charge of sending and receiving data from all of the hardware on board the system, and processing it into the proper formats for transmission. This section will first provide a general routine of the code, and then discuss how it interfaces with each of the different hardware.

General Overview

To ensure that the system communicates with MATLAB properly, a handshake with MATLAB is implemented before the ATSAM4S4B begins sampling from the system. Thus, upon startup, after properly initializing all of the hardware on board, the system waits for a message over UART sent by MATLAB. Once this message is sent, then the system samples an accelerometer input. This acceleration value is then sent to the FPGA in order to be filtered, using an SPI interface. The system waits for the FPGA to finish filtering, and receives the filtered data. Finally, the system sends this data into MATLAB, before entering a delay. This delay is implemented in order to ensure that the system samples at the appropriate rate (i.e., 400 Hz).

Thus, the microcontroller routine can be broken down into the following steps:

1. Initialize
2. Handshake
3. Sample
4. Filter
5. Transmit
6. Delay

1. Initialize

Apart from the standard initialization routine for the ATSAM, additional modules must be initialized in order for this microcontroller to work. The system makes use of the UART module, the SPI module, and the Timer/Counter modules. The UART and SPI must be set at the appropriate clock rates in this stage, which can be seen in Appendix A. The clock rates associated with these values are discussed in Section 4 and 5 in more depth.

2. Handshake

This section serves to ensure the transmission occurs smoothly. The order in which bits are transmitted is significant, and this will be discussed further in the Transmit section. This stage hangs the entire process until MATLAB is ready to start receiving data, in which MATLAB will send a 1 over UART to the microcontroller. Afterwards, the rest of the process may proceed.

3. Sample

The output pins from the accelerometer are connected to the ADC pins, as shown in the top-level modules block diagram. The ADC module will sample the analog signal at a frequency of 400 Hz, and get rid of the DC offset of the accelerometer outputs by transforming the acceleration data from a floating number to a signed char (i.e., output ranges from -128~127). This transformation is done in microcontroller by first scaling the 0-3.3V inputs to a 0-255 value, and then subtract an offset of 128 from the value.

4. Filter

The system used the same SPI communication scheme from Lab 7, AES Encryption, in order to communicate with the FPGA. The only changes made were adjusting the data sizes in order to suit the project's needs. The char data type was sent from the microcontroller to the FPGA, and an int was received in exchange.

5. Transmit

Both the BlueSMiRF and the ATSAM4S4B were set to communicate at 19,200 bits per second, using a UART protocol. When rates were increased, the rate of transmission errors to MATLAB greatly increased due to a delay in MATLAB. This error was not critical to the design our project, future researchers who are seeking to improve upon our current system may investigate it further. This component of the project served two purposes: to create a handshake to initiate data transfer, and to transfer filtered data.

Transmission between MATLAB and the ATSAM is limited by the rate at which MATLAB can read the serial port that the Bluetooth module is connected to. The execution time of this function is 0.002 seconds, or approximately 2000 Hz. Adding additional functionality such as real-time plotting makes the execution time much longer (i.e., approximately 0.1s execution time), so MATLAB cannot plot data from the Bluetooth serial buffer at the same rate that it is being filled by the ATSAM. When the buffer reaches its full capacity, 512 bytes, then the BlueSMiRF automatically disconnects from MATLAB and ceases to receive data. Therefore, for long sample times or real-time plotting capabilities, MATLAB should be avoided. However, for the final project, MATLAB was used in order to demonstrate this system's capabilities.

The BlueSMiRF module can only send one byte at a time. Therefore, the order in which MATLAB receives these bytes in its buffer are of significance, because MATLAB is reading two bytes at a time as shorts. Therefore, the purpose of the handshake is to ensure that the data is being received in the correct order; MATLAB flushes its serial buffer before performing the handshake with the ATSAM, in order to ensure that the bytes it receives are received in the correct order.

6. Delay

The entire process takes place inside a while loop. In order to control the sampling frequency, the TC_CH[0] counter was used, which is a component of the timer/counter module in the microcontroller family. The following two lines of code was called in the beginning of the while loop. It resets the counter, and sets the comparison value to duration, which is currently defined as 2500 microseconds (the period for a 400 Hz signal).

```
// Code implemented at the beginning of while loop
TC0->TC_CH[0].TC_CCR.SWTRG = 1; // Reset counter
TC0->TC_CH[0].TC_RC = duration * 20; // Set compare value
```

In the end of the while loop, the following line of code was called, in order to delay until the desired duration, 2500ms, has been reached.

```
// Code implemented at the end of while loop
while(!(TC0->TC_CH[0].TC_SR.CPCS)); // Wait until an RC Compare has occurred
```

FPGA Design

General Overview

The FPGA hardware can be split up into two general routines: SPI and convolution. The purpose of the SPI routine is to receive data from the ATSAM, and the purpose of the convolution routine is to perform a digital convolution in order to acquire filtered signals.

SPI

This module receives and transmits data from the FPGA to the ATSAM. It expects a signed char as input, and a short as output. The ATSAM generates the serial clock that controls this module, which is performed at 20 kHz.

Convolution

The FPGA currently implements a convolution by breaking down the process into 153 cycles, for each of the 153 coefficients of the band stop filter.The process starts immediately after load, a signal from the ATSAM, is de-asserted. A counter, which controls a signal called *round*, is used to count the number of "rounds", or cycles that the band-stop filter has undergone. At each round, a filter coefficient and past input are chosen from the modules *filterCoeffs* and *pastData*, respectively.

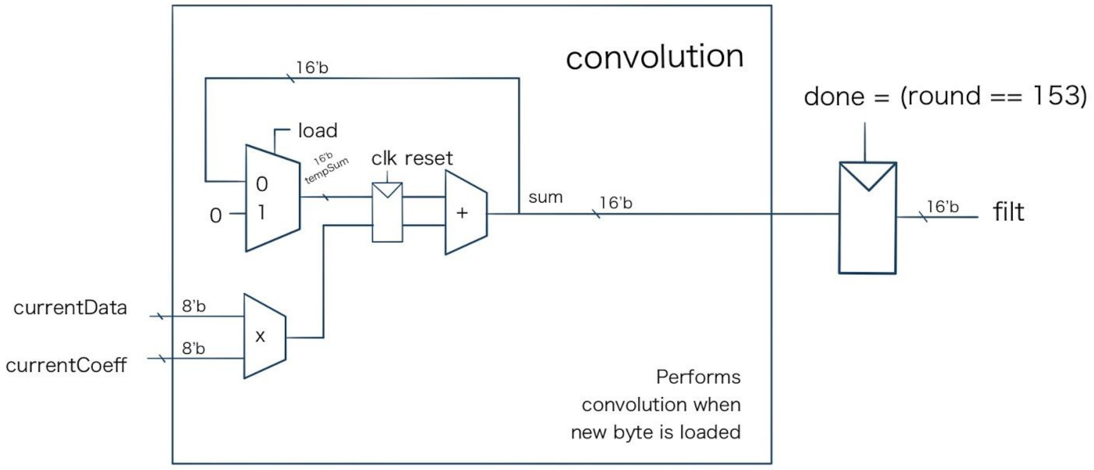


Figure 5, A Hardware Diagram of Convolution.

The module *filterCoeff* is implemented as a ROM, which chooses the coefficient corresponding to the round that the filter is in. The module *pastData* is used to store previous inputs into the FPGA filter. These data are stored in a series of 153 registers, and are selected via a multiplexer that takes in these 153 past inputs. The multiplexer is controlled by the value of *round.* The implementation of these modules can be seen in more detail in Appendix B. These chosen signals become currentData and currentCoeff.

The two signals, currentData and currentCoeff, two signals are multiplied together, then added to a recurring sum. Once 153 rounds have been completed, then the recurring sum is the filtered value. This value is a short data type, and is sent to the SPI module to be transmitted.

Results

The project met the proposed specifications in the project proposal. It is able to sample data at a rate of 400 Hz, and filter out 60 Hz signals using a finite impulse response filter implemented in hardware. In addition, it has demonstrated the ability to transmit data three meters away from a recipient device.

Challenges

The most time spent on the project was spent resolving the timing issues in communicating between the BlueSMiRF module and MATLAB. MATLAB's code does not execute quickly, so it cannot easily read values in real-time -- the Bluetooth serial buffer fills to capacity, and the BlueSMiRF module automatically disconnects once this occurs. In addition, detailed outlines of MATLAB's Bluetooth modules are black-boxed, so tracking down implementation specific bugs is difficult to acquire without carefully reading through the code.

Another significant part of the project was spent verifying that the band-stop filter functioned according to specifications. Some challenges encountered in this module included mistimed control signals and incorrect FIR filter coefficients. A testbench proved to be essential in debugging the FPGA.

Differences between Proposed and Current Implementation

The current system is able to sample data at the specified 400 Hz rate, and the band stop filter is able to figure out 60 Hz signals, as specified in the proposal. However, real-time plotting of data is not implemented because of the BlueSMiRF module's timeout issues, as discussed above. Instead, 1000 data points are sampled in order to collect some data while avoiding the timeout issues.

Sampling Rate Validation

In order to investigate the validity of our current filter, we used the oscilloscope wave generator to generate sinusoidal waves at different frequencies. Figure 6 is a screenshot from the oscilloscope for the 60 Hz wave. For all inputs sinusoidal signals, the peak to peak amplitude was 388mV, and the offset was 1.6 V. These voltage signals were directly inputted into an ADC pin, where they are sampled by the ADC module at 400 Hz.

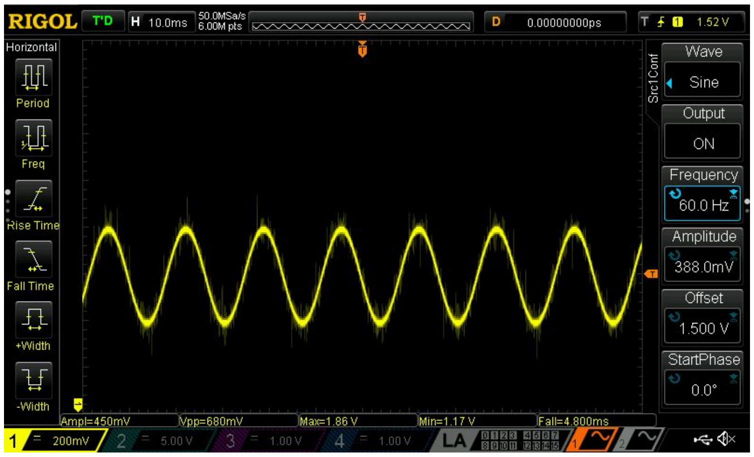


Figure 6, Oscilloscope Wave Generator of a 60 Hz Sinusoidal Signal

The filtered input was transmitted to Matlab and was saved in the Matlab workspace. Below is a plot of the scaled 40 Hz signal. The sampling frequency can be validated by comparing the measured frequency to the actual signal frequency. As shown in Figure 7, The measured frequency is 400 Hz.

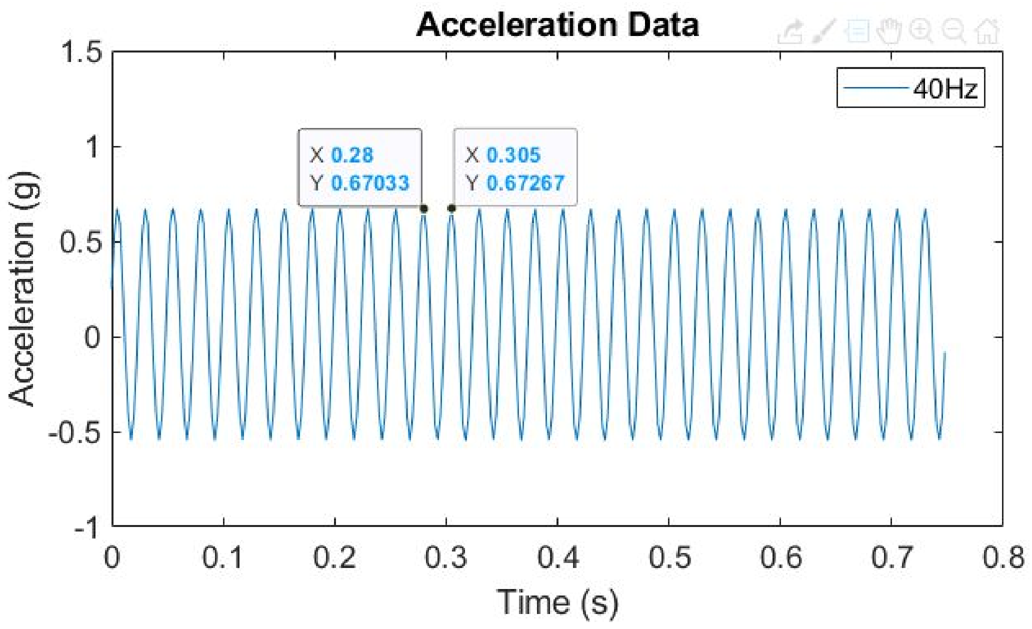


Figure 7, Matlab Plot for Filtered, Scaled 40 Hz Signal.

Filter Validation

Experimental data was gathered to ensure that the filter functioned as specified. Sinusoidal signals were generated using the RIGOL MSO1104 Oscilloscope, at varying frequencies. An example of the signal outputs of a 60 Hz and a 40 Hz signal is shown below.

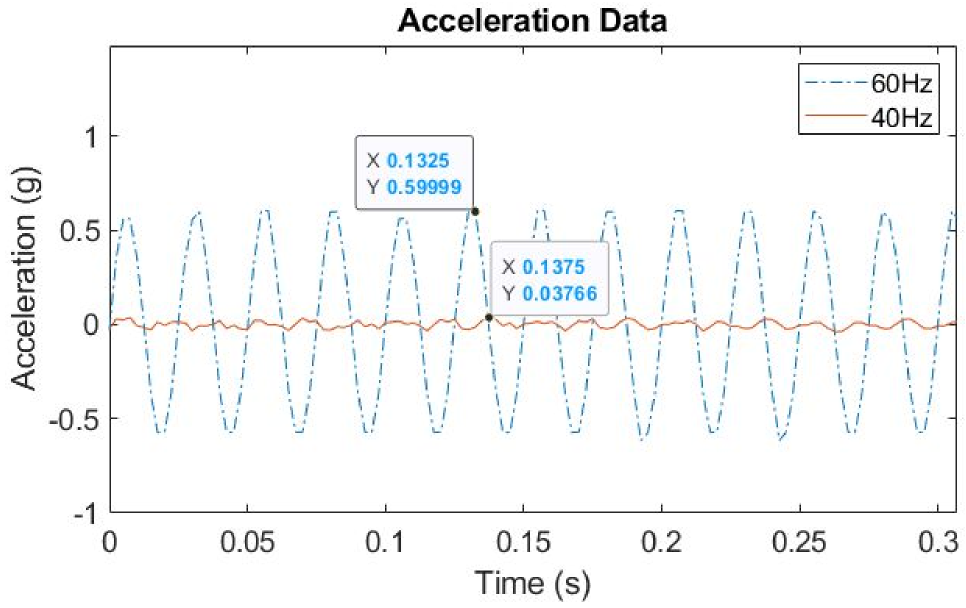


Figure 8, Magnitude Comparison of Filter Inputs at 60Hz and 40Hz.

Taking the ratio of the amplitudes of the 40 Hz signal to the 60 Hz signal, we see a gain of 6.5. Therefore, the magnitude difference between the signals is approximately 16 dB. Additionally, a more comprehensive characterization of this filter was performed. The results of this characterization are shown in Figure 8, which agree closely with the theoretical plot specified in Figure 2.

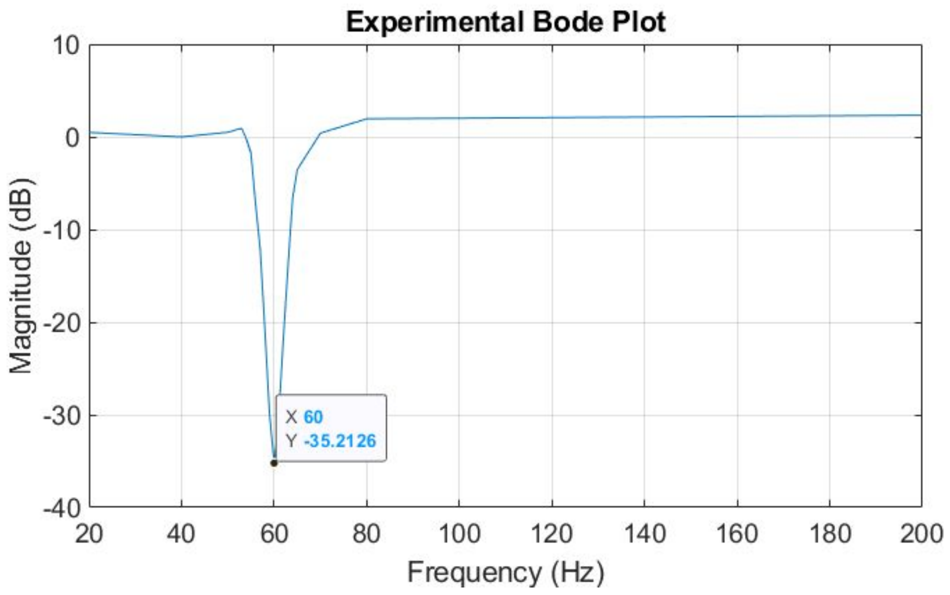


Figure 9, Experimental Magnitude Bode Plot.

References

[1] Bluetooth Data Module Command Reference & Advanced Information User's Guide, https://cdn.sparkfun.com/datasheets/Wireless/Bluetooth/bluetooth_cr_UG-v1.0r.pdf

[2] Small, Low Power, 3-Axis ±3 g Accelerometer, https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf

[3] Implementing a Low-Pass Filter on FPGA with Verilog, https://www.allaboutcircuits.com/technical-articles/implementing-a-low-pass-filter-on-fpga-with-verilog/

Parts List

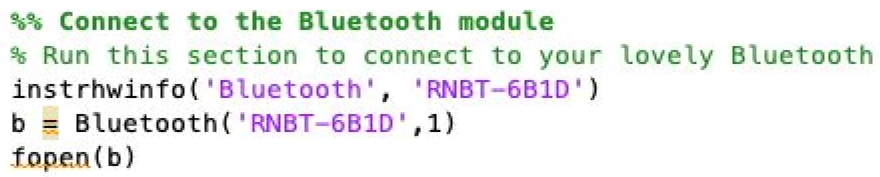
Part	**Source**	**Vendor Part #**	**Price (including tax)**
Bluetooth Modem - BlueSMiRF Silver	Amazon / 3DMakerWorld, Inc.	B004GTUI76	$32.42
ADXL335 Triple-Axis Accelerometer	Amazon / Adafruit	B00NAY2TUC	$15.19

Appendix A: For Future Students
How to Interface the BlueSMiRF with Matlab

Matlab has a Bluetooth toolbox that can be used to connect to Bluetooth devices. It is able to receive ASCII and binary data, and is able to interpret binary data as integers or signed/unsigned chars.

1. How to Connect to Bluetooth

After connecting your Bluetooth device to your laptop, Bluetooth can be connected to Matlab using the following command:



```
%% Connect to the Bluetooth module
% Run this section to connect to your lovely Bluetooth
instrhwinfo('Bluetooth', 'RNBT-6B1D')
b = Bluetooth('RNBT-6B1D',1)
fopen(b)
```

The Matlab instruction "instrhwinfo" returns the instrumentation details (i.e., data bytes sent, data bytes received, etc.). The first field specifies the device type, and the second field specifies the device name. The second line of code saves the device to the Matlab workspace, and can be later referred back to to open or close the device. After using the command, "fopen(b)", your device should now be connected to the Matlab.

2. How to Configure Bluetooth

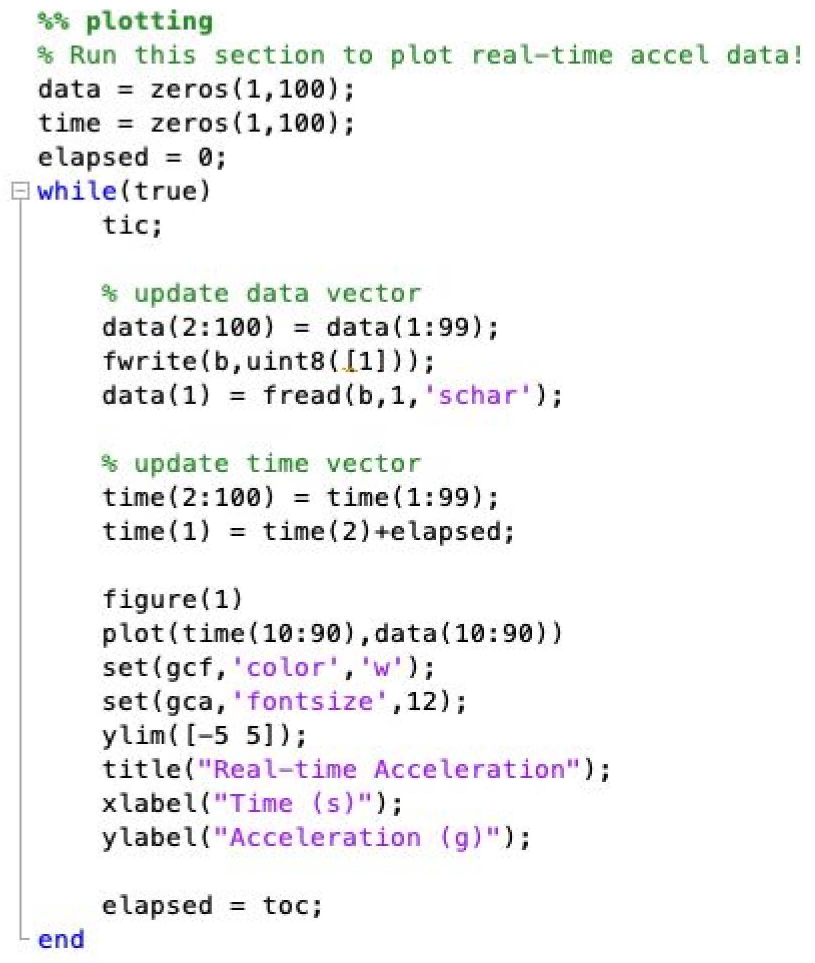
To configure Bluetooth, refer to the RN-42 datasheet, which is listed as Reference [1] in this report.

3. How to Write/Receive Data from Bluetooth

Data is typically received using the *fread* function. For example, you can type "fread(b,1,'schar')" to receive 1 signed character from your Bluetooth device named "b". Similarly, you can write to the Bluetooth device using the *fwrite* function. For example, "fwrite(b,uint8([1]))" would write the number "1" to your bluetooth device named "b" as an unsigned 8-bit binary number. For more details, please refer to the *fread* and *fwrite* page in mathworks.com.

4. How to Plot Data in Realtime

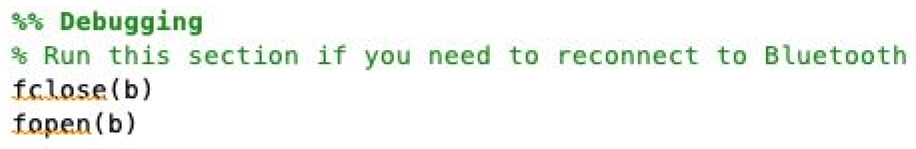
Although not commonly used, Matlab can program while loops. Below is a sample code of how you may plot the most recent 100 data points in Matlab. As a side note, the plot function may cause a delay in data transmission, and hence if you decide to plot in real time, please expect to see some delay/slowdown in data transmission.



```
%% plotting
% Run this section to plot real-time accel data!
data = zeros(1,100);
time = zeros(1,100);
elapsed = 0;
while(true)
    tic;

    % update data vector
    data(2:100) = data(1:99);
    fwrite(b,uint8([1]));
    data(1) = fread(b,1,'schar');

    % update time vector
    time(2:100) = time(1:99);
    time(1) = time(2)+elapsed;

    figure(1)
    plot(time(10:90),data(10:90))
    set(gcf,'color','w');
    set(gca,'fontsize',12);
    ylim([-5 5]);
    title("Real-time Acceleration");
    xlabel("Time (s)");
    ylabel("Acceleration (g)");

    elapsed = toc;
end
```

5. How to Restart Bluetooth

For debugging purposes, you may want to close and reopen your Bluetooth if it is not responding. Use the following code to close and reopen your device.



```
%% Debugging
% Run this section if you need to reconnect to Bluetooth
fclose(b)
fopen(b)
```

Appendix B: Keil - C Code

```
/*
Andrew M. Pham & Dana ShangGuan
December 11, 2019
ampham@g.hmc.edu, dshangguan@g.hmc.edu

Top level routine for the ATSAM.
This will take in data from an accelerometer, send it to an FPGA to filter in real time,
send the data to a Bluetooth module connected to a computer.

*/

#include "SAM4S4BDeviceDriver/SAM4S4B.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define XPIN 0
#define YPIN 1
#define ZPIN 2

#define LOAD_PIN    5
#define DONE_PIN    6

void accelInit(int, int);
signed char accelRead(int);
void sendChars(char, char*);
void initialize(void);
void tcDelayStart(uint32_t duration);
void tcDelayEnd(void);

int main(void){
              // initialize variables
              signed char xchar = 0;
              char filtx[2];
              int i;

              // initialize all modules
              initialize();

              // handshake! Start sending data after Matlab is ready
              while(!uartRx())

    while(1){
  /**********************
 A Tc counter function that tracks the time
 at the beginning of this cycle.
  **********************/
       tcDelayStart(2500);
/***********************
ADC Module: Accelerometer code
************************/
       // read from accelerometer x-axis
       xchar = accelRead(XPIN);

/************************
SPI Module: Send data to FPGA for filtering
************************/
        sendChars(xchar, filtx);

/**********************
 BlueSMiRF/UART Module
Send the filtered values over Bluetooth to be stored on a computer.
Note that the values of the int should be restricted from -128 to 127, because
the original acceleration inputs into the system should not exceed these values,
and the filtering should only attenuate the signal, not gain it.
**********************/

       // send filtered data
       for(i = 0; i<2; i++){
              // change char order according to MATLAB standard
              uartTx(filtx[1-i]);
       }

        // wait until 2500 microseconds have passed,
                            // which gives us a sampling rate of 400Hz.
                            tcDelayEnd();
  }
}


// initialize all modules
void initialize() {
    samInit();
              pioInit();
              tcDelayInit();
    accelInit(1, 1); // set gain, offset
              uartInit(0,123); // set the baud rate for Bluetooth transmission to be 19.kHz

              // SPI and band-stop filter
              // setting up SPI for data transmission
              // set SCK to be much higher than sampling frequency
              spiInit(MCK_FREQ/2000, 0, 1);

    // initialize load and done pins
    pioPinMode(LOAD_PIN, PIO_OUTPUT);
    pioPinMode(DONE_PIN, PIO_INPUT);
}


// initialize the accelerometer and ADC pins
void accelInit(int gain, int offset) {
    adcInit(8); // Set 8 bit resolution on the ADC module
    adcChannelInit(XPIN, gain, offset);
    adcChannelInit(YPIN, gain, offset);
    adcChannelInit(ZPIN, gain, offset);
}


// read from desired ADC channel
signed char accelRead(int channel) {
              // read from accelerometer and convert
    float x = adcRead(channel);
              signed char xchar = round((x-1.61)/(3.3/255));
              return xchar;
}


// send chars to and read from FPGA
void sendChars(char xchar, char *filtx) {
              /***********************
              SPI
              ***********************/
              int i;

              pioDigitalWrite(LOAD_PIN, 1);
              spiSendReceive(xchar);
              pioDigitalWrite(LOAD_PIN, 0);

              // wait for FPGA to finish processing stuff
              while(!pioDigitalRead(DONE_PIN));

              // get filtered data
              if (pioDigitalRead(DONE_PIN)) {
                     for(i = 0; i < 2; i++){
                            filtx[i] = spiSendReceive(0);
                     }
              }
}


// start the counter in the beginning of while loop
void tcDelayStart(uint32_t duration) {
    TC0->TC_CH[0].TC_CCR.SWTRG = 1; // Reset counter
    TC0->TC_CH[0].TC_RC = duration * 20; // Set compare value
}


// delay for desired duration in the end of while loop
void tcDelayEnd() {
    while(!(TC0->TC_CH[0].TC_SR.CPCS)); // Wait until an RC Compare has occurred
}
```

Appendix C: Quartus - System Verilog Code

```
/////////////////////////////////////////////
// top.sv
// HMC E155 December 11, 2019
// ampham@g.hmc.edu, dshangguan@g.hmc.edu
/////////////////////////////////////////////

/***************
testbench

test band-stop filter
****************/

module testbench();
    logic clk, load, done, sck, sdi, sdo, reset;
        logic [31:0] filtData;
    logic [7:0] i, currentTransmit;
        logic [31:0] expected[0:600]; // I just want the values to compare
        logic [7:0] inputSignal[0:300]; // Where our data will be loaded
        logic [31:0] vectornum;
        logic [11:0] answer;

    // device under test
    top filt(clk, reset, sck, sdi, sdo, load, done);

    // test case
    initial begin
      $readmemh("inputSignal.txt", inputSignal);
      $readmemh("outputSignal.txt", expected);
            vectornum = 0;
    end

    // generate clock and load signals
    initial
        forever begin
            clk = 1'b0; #5;
            clk = 1'b1; #5;
        end

    initial begin
      i = 0;
            currentTransmit = inputSignal[0];
            reset = 1; #10;
      load = 1'b1;

    end

      //      shift in test vectors, wait until done, and shift out result
    always @(posedge clk) begin
                  if (i == 8) begin load = 1'b0;reset = 0;end
                  if (i<8) begin
```

```
                        #1; sdi = currentTransmit[7-i];
                        #1; sck = 1; #5; sck = 0;
                        i = i + 1;
                      end else if (done && i < 40) begin
                        #1; sck = 1;
                        #1; filtData[39-i] = sdo;
                        #4; sck = 0;
                        i = i + 1;
                      end else if (i == 40) begin
                                    load = 1'b1;
                                    i = 0;
                                    vectornum = vectornum+1;
                                    currentTransmit = inputSignal[vectornum];
                                    answer  = expected[vectornum];
                      end
    end

endmodule



/***************
top

The top level module that controls the communication between spi and FPGA
****************/

module top(input logic clk, reset,
            input logic sck,
            input logic sdi,
            output logic sdo,
            input logic load,
            output logic done);


        logic [7:0] data;
    logic [7:0] round;
    logic[31:0] filt;


        counter c0(clk, load, round, done);
    spi spi0(sck, sdi, sdo, done, data, filt);
    convolution filt0(clk, reset, load, done, round, data, filt);

endmodule



/***************
spi

load newdata and get filtered data
****************/
```

```
module spi(input  logic sck,
               input  logic sdi,
               output logic sdo,
               input  logic done,
               output logic [7:0] data,
               input  logic [15:0] filt);

    logic        sdodelayed, wasdone;
    logic [15:0] filtCap;

    // assert load
    // apply 24 sclks to shift in key and plaintext, starting with plaintext[0]
    // then deassert load, wait until done
    // then apply 32 sclks to shift out cyphertext, starting with cyphertext[0]

    // rewrite to only read data of 1 axis
    always_ff @(posedge sck)
        if (!wasdone)  {filtCap, data} = {filt, data[6:0], sdi};
        else           {filtCap, data} = {filtCap[14:0], data, sdi};

    // sdo should change on the negative edge of sck
    always_ff @(negedge sck) begin
        wasdone = done;
        sdodelayed = filtCap[14];
    end

    // when done is first asserted, shift out msb before clock edge
    assign sdo = (done & !wasdone) ? filt[15] : sdodelayed;
endmodule


/***************
counter

keep track of the current coeff and data indexs
****************/

module counter( input logic clk,
                    input logic load,
                    output logic [7:0] round,
                    output logic done);

    always_ff @(posedge clk)
        if (load)     round <= 0;
        else if (!load && round < 153) round <= (round + 1);

    // Only asserts once all 305 taps have been added in the convolution.

        always_comb begin
              if (round == 153 && !load) done <=1;
              else done <=0;
        end

endmodule
```

```
/**************
convolution

filters out 60 Hz noise
*************/

module convolution(input logic clk, reset, load, done,
                                            input logic [7:0] round,
                   input logic signed [7:0] newData,
                   output logic signed [15:0] sum);

    logic [15:0] tempSum;
    logic signed [15:0] currentData, currentCoeff;



    pastData pd0(clk, load, reset, newData, round, currentData[7:0]);
    filterCoeff coeff0(round, currentCoeff[7:0]);


        always_comb begin
              if (currentData[7]) currentData[15:8] <= 8'b11111111; // sign-extension
              else currentData[15:8] <= 8'b0;
              if (currentCoeff[7]) currentCoeff[15:8] <= 8'b11111111; // sign-extension
              else currentCoeff[15:8] <= 8'b0;
        end

    always_ff @(posedge clk)
      if (load) tempSum <= 0;
      else if (!done && round != 0) tempSum <= tempSum + currentData*currentCoeff;
      else sum = tempSum;

endmodule



/**************
pastData

loadData is data that has been loaded through the SPI.
round is a counter indicating the stage of the convolution.
currentData is the "current" data needed for that stage in convolution.

*************/

module pastData(input logic clk, load, reset,
                input logic [7:0] newData,
                input logic [7:0] round,
                output logic [7:0] currentData);

  int i;
  logic [7:0] storedData[0:152];

  always_ff @(posedge clk) begin
```

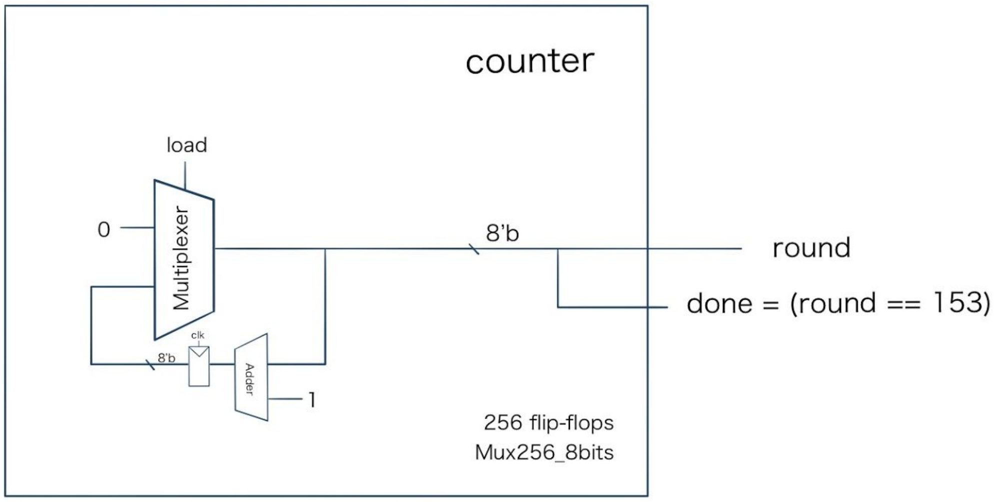
```
      if (reset)
              begin
                     i = 0;
                     for(i=0; i<153; i=i+1)
                            storedData[i] <= 0;
              end
      else if (round == 0 && load == 0)
        begin
          storedData[1:152] <= storedData[0:151];
          storedData[0] <= newData;
        end
       end

       always_comb
              if(round != 0) currentData = storedData[round-1];
              else currentData = 0;

endmodule



/***************
filterCoefficients

The input is the convolution round number.
Outputs coefficients for the FIR band stop filter.
Note that all coefficients except for coefficients[152] are scaled,
in order to become whole numbers.

****************/

module filterCoeff(input  logic [7:0] round,
            output logic [7:0] coeff);

  // coefficients implemented as a ROM
  logic [7:0] coefficients[0:152];

  initial   $readmemh("filterCoeff400flip.txt", coefficients);
  always_comb
              if(round != 0) coeff = coefficients[round-1];
              else coeff = 0;

endmodule
```

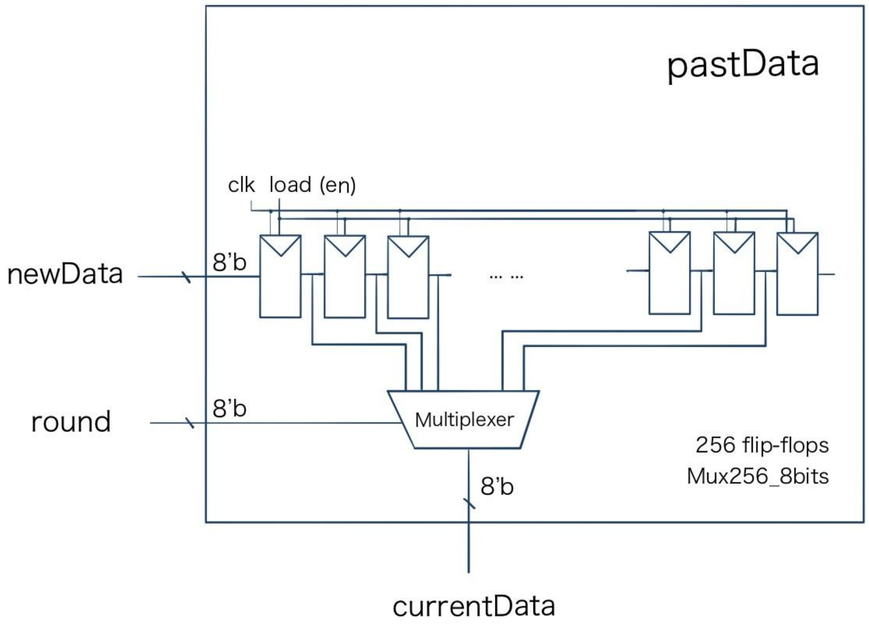
Appendix D: MATLAB Code

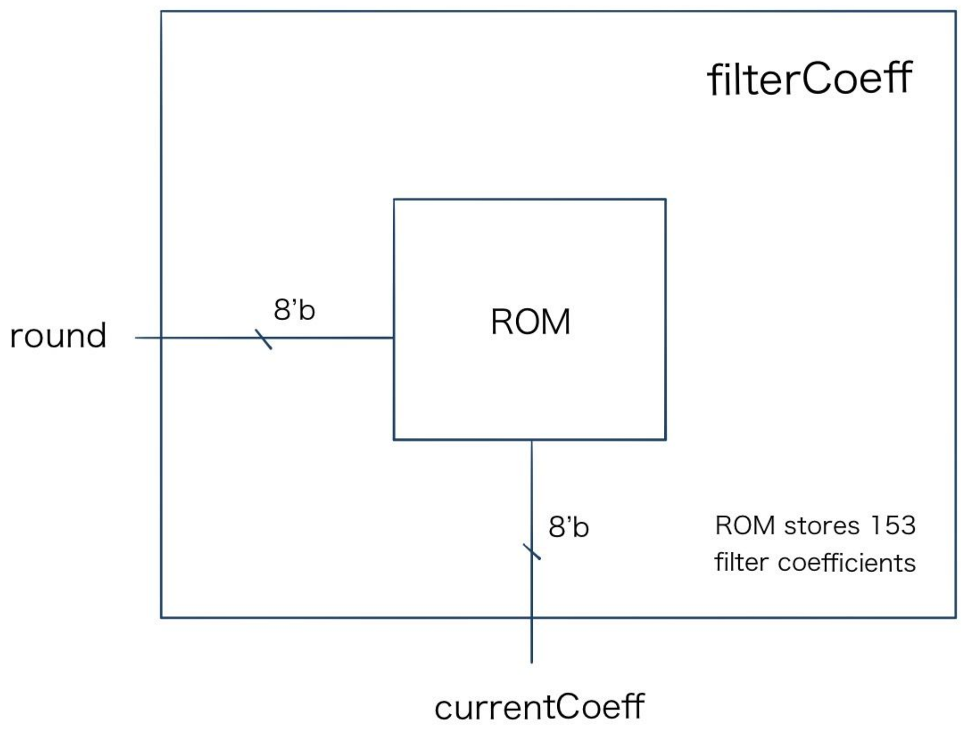
```
%% Connect to the Bluetooth module
% Run this section to connect to our lovely Bluetooth

instrhwinfo('Bluetooth', 'RNBT-6B1D')
b = Bluetooth('RNBT-6B1D',1)
fopen(b)


%% Demo: sample at 400 Hz

data200 = zeros(1,1e3);
fclose(b);
flushinput(b);
fopen(b);
fwrite(b,1);

while (abs(max(fread(b,5,'short'))) > 3000)

    fclose(b);
    flushinput(b);
    fopen(b);

end

fwrite(b,1);
for i = 1:length(data200)

    % update data vector
    data200(i) = fread(b,1,'short');

end
fclose(b);

% plotting
scale = 131.6062; % from scaling coefficients
acceleration = data200.*(3.3/250)/0.3/scale;
time = 0:0.0025:2.4975;
a = 300;
plot(time(1:a), acceleration(200:a+199)-mean(acceleration(200:a+199)))
hold on
set(gcf,'color','w');
set(gca,'fontsize',14);
ylim([-1.0 1.5])
title("Acceleration Data");
xlabel("Time (s)");
ylabel("Acceleration (g)");

%%
bode20 = -20*log(max(data40)/max(data20));
bode40 = -20*log(max(data40)/max(data40));
```

```
bode50 = -20*log(max(data40)/max(data50));
bode53 = -20*log(max(data40)/max(data53));
bode54 = -20*log(max(data40)/max(data54));
bode55 = -20*log(max(data40)/max(data55));
bode56 = -20*log(max(data40)/max(data56));
bode57 = -20*log(max(data40)/max(data57));
bode58 = -20*log(max(data40)/max(data58));
bode59 = -20*log(max(data40)/max(data59));

bode60 = -20*log(max(data40)/max(data60));
bode61 = -20*log(max(data40)/max(data61));
bode62 = -20*log(max(data40)/max(data62));
bode63 = -20*log(max(data40)/max(data63));
bode64 = -20*log(max(data40)/max(data64));
bode65 = -20*log(max(data40)/max(data65));
bode70 = -20*log(max(data40)/max(data70));
bode80 = -20*log(max(data40)/max(data80));
bode200 = -20*log(max(data40)/max(data200));

mag = [bode20,bode40,bode50,bode53,bode54,bode55,bode56,bode57,bode58,...
     bode59,bode60,bode61,bode62,bode63,bode64,bode65,bode70,bode80,bode200];
freq = [20,40,50,53,54,55,56,57,58,59,60,61,62,63,64,65,70,80,200];

plot(freq,mag)
grid on
set(gcf,'color','w');
set(gca,'fontsize',14);
title("Experimental Bode Plot");
xlabel("Frequency (Hz)");
ylabel("Magnitude (dB)");


%% plotting in real time
% Run this section to plot real-time accel data!

data200 = zeros(1,1000);
time = zeros(1,1000);
%scale = 130.28; %1000 Hz sampling
scale = 131.6062;
elapsed = 0;

fclose(b);
flushinput(b);
fopen(b);
fwrite(b,1);
while(true)
    tic;

    % update data vector
    data200(2:1000) = data200(1:999);
    data200(1) = fread(b,1,'short')*(3.3/250)/0.3/scale;
    % update time vector
    time(2:1000) = time(1:999);
    time(1) = time(2)+elapsed;

    figure(1)
    plot(time(200:1000),data200(200:1000))
%     set(gcf,'color','w');
%     set(gca,'fontsize',12);
%     ylim([-10 10])
%     title("Real-time Acceleration");
%     xlabel("Time (s)");
%     ylabel("Acceleration (g)");

    elapsed = toc;
    %wait = 1 - elapsed;
    %pause(wait)
end
```

Appendix E: Block Diagrams for Filter Submodules






filterCoeff
round
8’b
ROM
8’b
ROM stores 153 filter coefficients
currentCoeff