Extracting Electrocardiographic Heart Rate Via FPGA

Final Project Report
December 9, 2017
E155

Benjamin Iten and Brenden Brown

Abstract:
Tracking a patient's biosignals is critical for monitoring the status of a patient in critical condition. Heart rate is one of the signals that can be accessed in a non-invasive manner. Other than pulse oximetry, electrocardiography is the main way to accomplish this. This project created a prototype of a heart rate monitor using an electrocardiographic setup (capacitive probes on the body wired to an amplification circuit). The amplified pulse was sent to an ADC, and then heart rate was extracted using time-domain peak finding, and sent serially to a Raspberry Pi. The Pi then displayed the result on a web page, accessible from the Internet.

Introduction

Electrocardiography is the noninvasive process of recording electrical activity from the heart using electrodes placed on the skin. Typically, this electrical activity is used to create an Electrocardiogram and extract heart rate. Our project accomplished the latter. Using a simplified electrode setup (3 electrodes as opposed to the 6-8 used in hospitals), we combined analog circuitry and FPGA data processing to extract a user's heart rate. This heart rate was sent to a Raspberry Pi and displayed on a webpage.

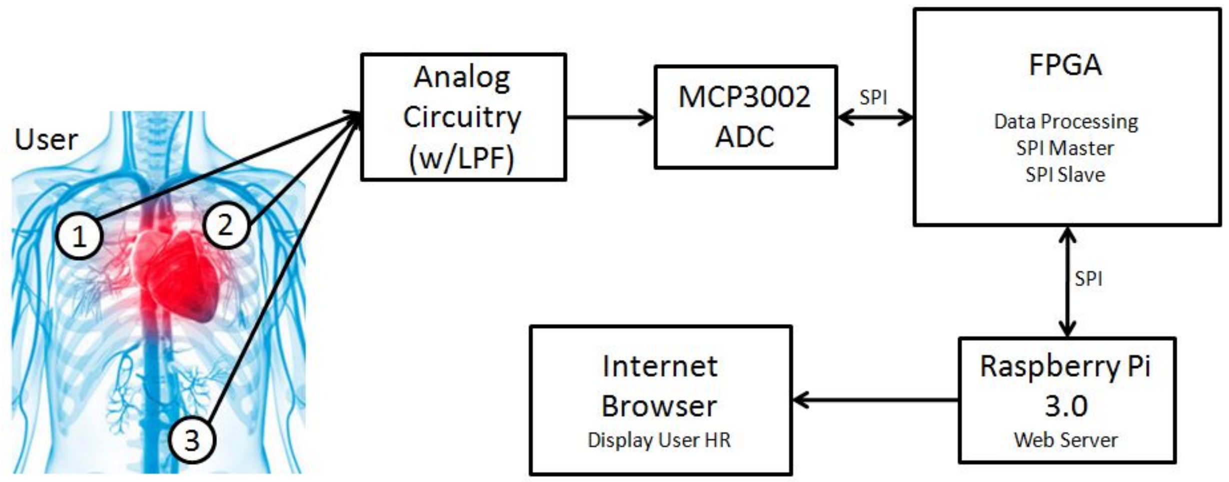


Figure 1. Overall System Block Diagram.

New Hardware

Two new pieces of hardware were used in the analog front end of the electrocardiographic signal processing. The analog hardware had three stages with the first stage being an instrumentation amplifier utilizing the AD623AN and the third stage being a non-inverting amplifier utilizing a MCP601 operational amplifier. The second stage was a first order low pass filter using passive components.

The instrumentation amplifier amplified the raw electrical pulse signals received from the probes. The low pass filter output a cleaner signal and then the third stage amplifier amplified the cleaned up signal once more before it was sent to the ADC. The corner frequency of the low pass filter was around 320 Hz and was determined through trial and error.

These analog components were selected with the constraints of being able to run off the Pi 5V power supply, a 3.3V power rail, and GND. The Pi 5V and GND provided the rails for the two ICs and the 3.3V was used to form a voltage divider to provide an adequate reference voltage for the instrumentation amplifier in the first stage. The instrumentation amplifier was chosen because it was able to provide a gain ranging from 1 to 1000 (we ended up using a gain of 51 to ensure we didn't hit the rails).

Three 3M capacitive probes attached via alligator clip cables were used to collect the raw electrical signal. Two of the probes were placed on either side of the heart and a third reference probe was placed on the abdomen and connected to the circuit ground.

Schematics

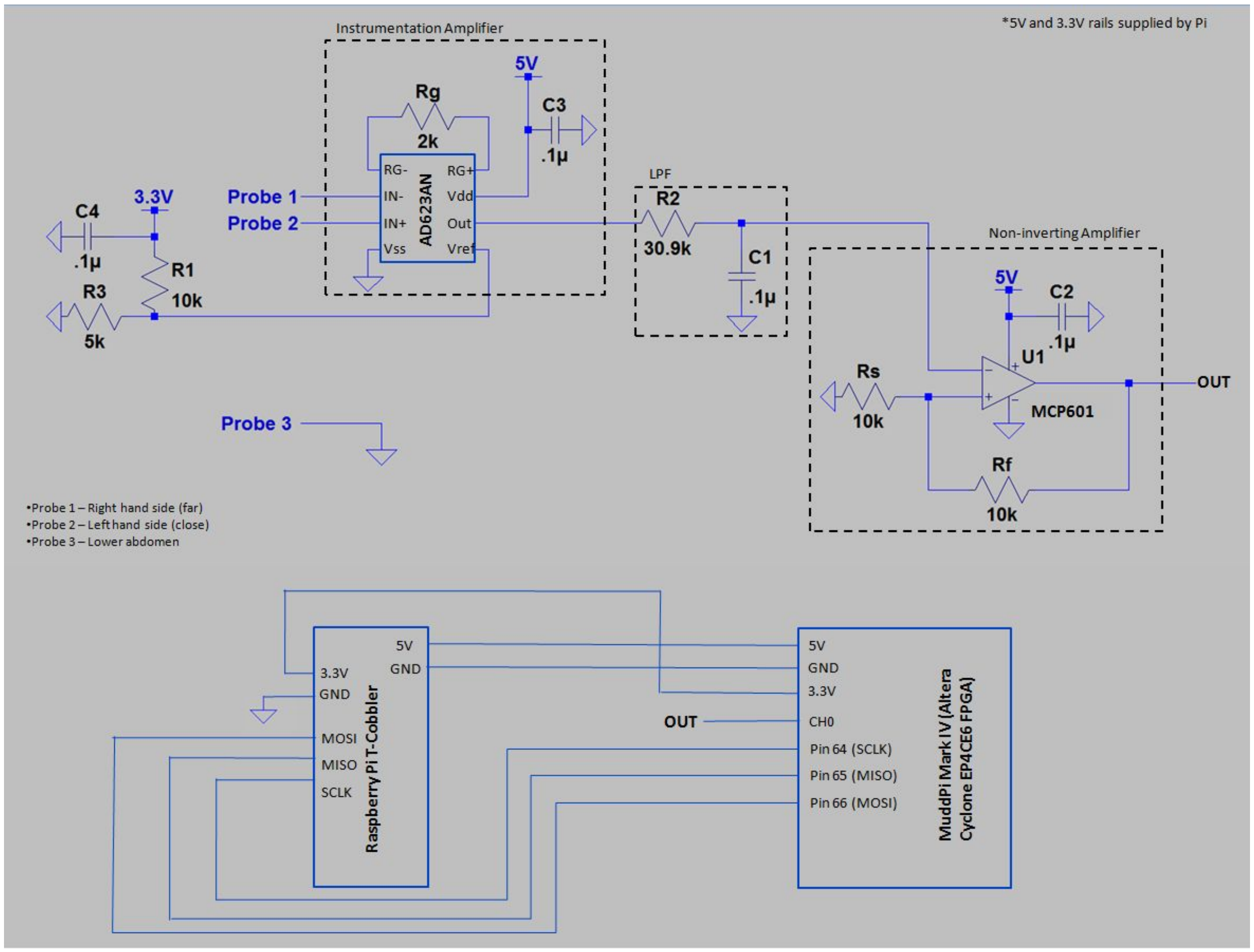


Figure 2. Overall System Circuit Schematic

Microcontroller Design

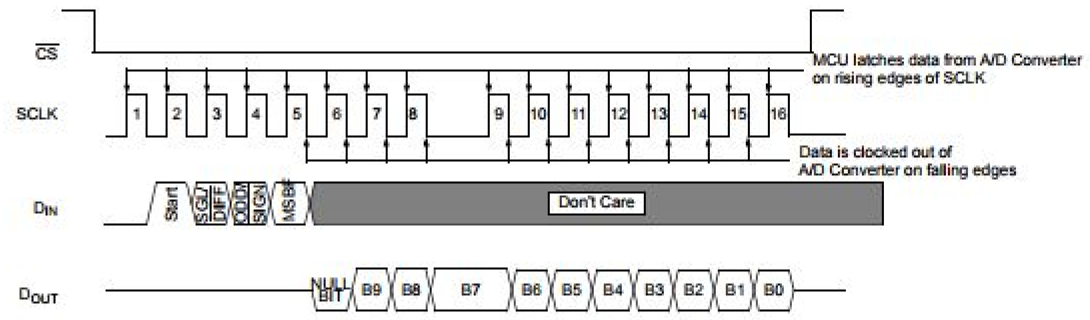
Data Processing:

The purpose of the Raspberry Pi is to send an SPI clock to the FPGA at roughly 200 kHz and generate a chip select signal to start communication with the FPGA. Instead of using a separate chip select wire, the PI and the FPGA begin communication when the MOSI bit goes high. This way, sixteen bits can be sent and received between the two devices. A one in the 8th bit place of the shift register will result in the FPGA returning the 8 bit HR result. The Pi will receive the 8 bit heart rate, convert it to a float value and run an HTML script to continuously display it on a webpage using C. HTML code on the Pi creates a visually appealing overall page and refreshes an iFrame with the webpage created from the C code for displaying the heart rate. The overall webpage page is refreshed every 5 seconds to display a real-time heart rate.

Simulation:
To test this portion of the system was working properly, the FPGA miso output was set to a fixed value and sent to the Pi. The webpage was opened and confirmed to displaying the floating value of the binary number sent by the FPGA. The FPGA also had a reset value, so to test that our page was updating in real-time, we would switch on the reset and see the webpage display change to the reset value, then switch off the reset and observe the webpage change back to the fixed miso value being sent.

FPGA Design

The purpose of the FPGA in this project was to act as an SPI slave and an SPI master as well as a time domain data processing instrument. The inputs to the FPGA are the on-board 40MHz clk, a PiSCLK, a mosi bit, a Din bit, and a reset bit. The reset bit is tied to one of the switches on the FPGA board, and this is helpful for debugging, and for starting in a known state. Mosi and PiSCLK are provided by the Raspberry Pi. The Din bit comes from the MCP3002 ADC, and gets connected to the Dout pin on the ADC. The outputs of the FPGA are SCLK, Dout, and CSBar which are sent to the ADC. A miso bit is also sent to the Pi.



$\overline{CS}$
MCU latches data from A/D Converter on rising edges of SCLK
SCLK
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Data is clocked out of A/D Converter on falling edges
DIN
Start
SGL/DIFF
ODD/SIGN
MSBF
Don't Care
DOUT
NULL BIT
B9
B8
B7
B6
B5
B4
B3
B2
B1
B0

Figure 3. SPI Communication Protocol for MCP3002.

First, the FPGA acts as an SPI master for communication with the MCP3002 10-bit analog to digital converter. A clock divider is used to take the on-board 40MHz clock, and divide it to around 250kHz for use as a serial clock. This SCLK signal is sent to the ADC. Using the serial clock, a second clock divide is performed to generate a chip select signal, CSbar, which is also sent to the ADC. When CSbar is set low, the ADC waits for a start, clock polarity, and channel select bit from the FPGA. On the negative edge of the CSbar signal, a shift register is loaded, and on each clock cycle the most significant bit is sent out and the bit received from the ADC is shifted in. In Figure 3 one can see how on the negative edge of the serial clock that the Din, Dout values change. The Din in the figure is the data into the ADC, which is referred to as Dout in the verilog. After 16 clock cycles, the 10 bit readings from the ADC is captured, and sent to the data processing module. Also, a ‘done’ bit is sent out of the SPI master module as well. This is used in another clock divider to set the sampling rate of the data processing module. The ~7800Hz refresh rate of the ADC value is slowed down to 400Hz for use in data processing.

There is also an SPI Slave module for communication between the FPGA and the Pi. The module takes in a serial clock from the Pi, PiSCLK, which operates at around 250kHz as well as a mosi bit. The Pi sends mosi high for one serial clock cycle, and at this, the shift register is loaded with the most recent heart rate value calculated, and on each successive clock cycle, bits are shifted and the most significant is sent the Pi on miso. After the heart rate bits are sent, the module will just output zeros.

The final module is the time domain, peak finding module: HRFind. This module takes in an ADC readings, as well as a newData flag, a reset, and it outputs the calculated heart rate. At the positive edge of the 400Hz newData signal, a data processing operation begins on the most recent ADC value. There first exists a reset state which, on reset, sets all of the inner signals to zero, and outputs heart rate zero.

The data processing algorithm works on a moving envelope which is calculated from 10, 200 bit shift registers. The sum of the 200 most recent samples is calculated, and then divided to get a moving average, which can be approximated at the baseline of the heart rate signal. On each new ADC readings, bit 0 - 9 are put into the first spot of the corresponding shift register (bit0Reg-bit9Reg), and each bit shifted out is combined into a subVal. The new ADC reading is added to movingAvg, and the subVal is subtracted. At setup, a baselineN variable tracks the number of points that have been added to the baselineSum. Once the number of points reaches 200, baselineN stays at 200 since movingAvg is calculated from the average of the sum of the most recent 200 points. Using this movingAvg, each new ADC value is compared to movingAvg plus a delta value. Delta is calculated based on what type of outputs are expected from the amplification circuitry. Based on what was seen in the lab delta was set to 2 (2 ADC ‘units’ which range from 0-1023). If a higher gain was achieved, a higher delta would be set. If the ADC value goes above movingAvg + delta, then a counter is started, and a startCount variable is set to one. Once the count exceeds around 200, and it goes above movingAvg + delta again, the count is captured into a variable, peakSep, and count is reset, and startCount is set to zero. Count is required to exceed 200 because this helps to avoid catching the same peak. The samples between peaks is relatively flat, but there is a sinc-looking peak at each heartbeat, and if the first bit above the envelope triggers the count start, then we know that we need to wait until the next peak, and avoid mistaking later points on the same beat as the next beat Peaksep is multipied by 2500 to convert the count to microseconds, stored in timeBetween, and then a hardware divide is performed where 1 minute in microseconds is divided by timeBetween. This gives a sampleHR, which is then sent out of the module.

To validate that the various modules worked, each was simulated in ModelSim with a test vector file. Once the modules passed the ModelSim stage, each was tested in the lab. First, the SPI master module was verified by using a phototransistor circuit hooked up to the ADC. On a flip of a switch, the FPGA would output to the onboard LEDs the most recent ADC reading in binary. After carefully analyzing Figure 3 to determine how the bit order is, the clock edge requirements, and making sure to have the Din of the module hooked up to Dout of the ADC, it was found to work. Next, the SPI slave module was tested by sending a constant number to the Pi. It was found that it was easier to operate the Pi shift register as sending a single 1 which would reset the FPGA shift register, as opposed to dealing with a chip select signal. This meant using the SPISendRecieve16 from the GPIO library with the input 0x0100 to receive the 8 HR bits from the FPGA in the last 8 bits of the register. Last, once the pipeline from the ADC to the Pi worked, it was verified that an generated pulse and sine wave would produce a heart rate reading. The width of the pulse was also varied, and the same heart rate was observed, showing that the minimum separation threshold worked. This validated the heart rate finding module.

Results

In the end, we were able to complete the data processing on the FPGA and extract a heart rate from the electrocardiographic setup. The data collected from the probes was very noisy despite the low pass filter and the supply rails of the Pi also had a lot of noise despite a healthy use of bypass capacitors. This led to certain instances of an inaccurate heart rate. However, based on our simulated data, we are confident that with a cleaner power supply and better probes, a more consistent heart rate display will result.

The most difficult parts of the project were in extracting a clean heart rate signal using the probes, and in finding the best practice to capture a heart rate from said noisy signal. There is a tradeoff between having a robust algorithm, and capturing a meaningful and accurate signal.

From an analog standpoint, a larger gain may have helped with produce a cleaner signal; one where the peaks were far more significant than the noise. This would have helped improve the accuracy of our peak finding algorithm.

The data processing algorithm could have been improved by using a more robust envelope, as well as a dynamic delta for pulse values. Also, it would be preferable to stabilize the heart rate output as the average of the most recent heart rates calculated to avoid large swings in calculated values that occurred during testing. It would also be better to perform peak separation based on the maximum of each cycle, as opposed to the first value which goes out of the envelope. This would lead to a more accurate heart rate, since each beat is not identical.

References

Digital Design and Computer Architecture, Sarah L. Harris & David Money Harris

Datasheets:

- MCP601 http://ww1.microchip.com/downloads/en/DeviceDoc/21314g.pdf
- AD623AN http://www.analog.com/media/en/technical-documentation/data-sheets/AD623.pdf

Design References:

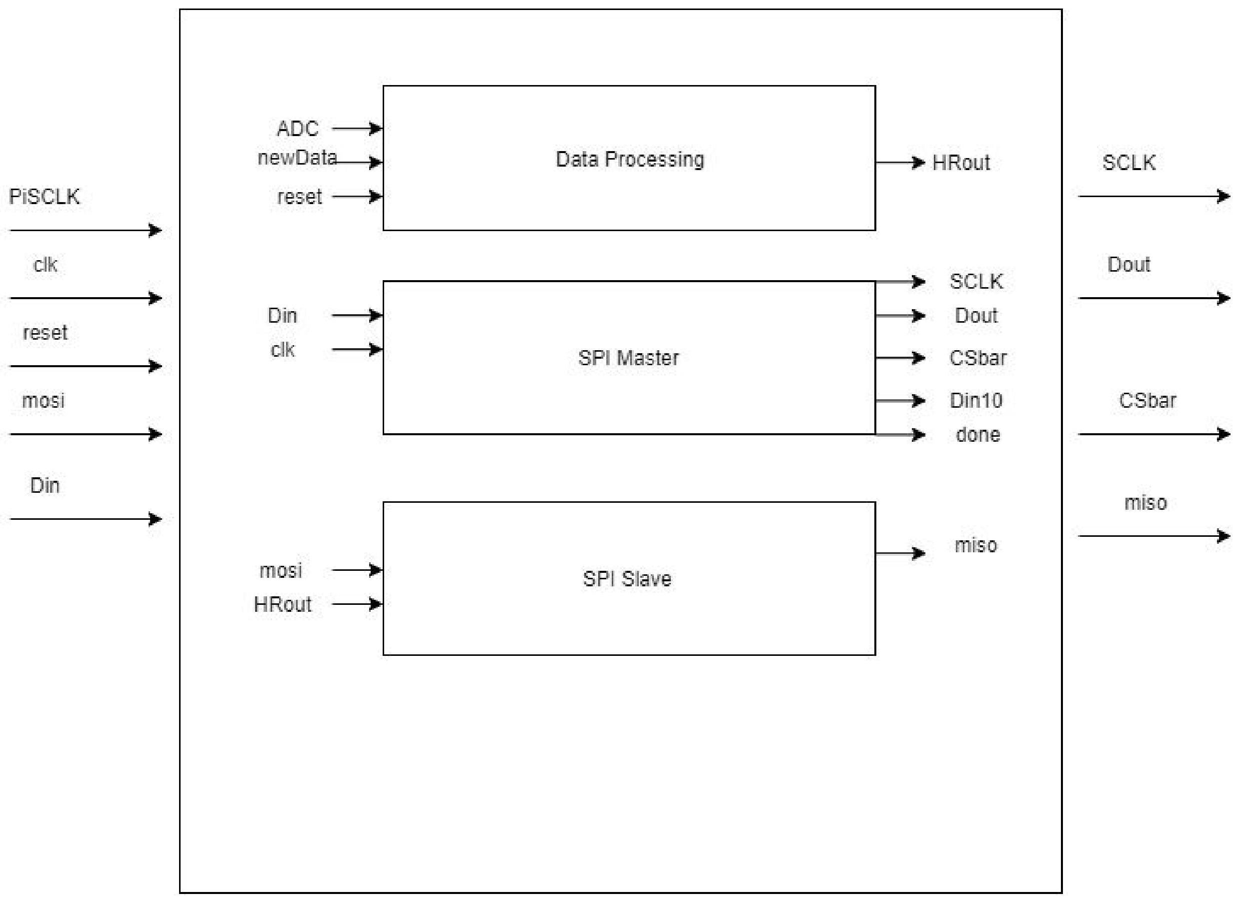
- E84 Spring 2017, Lab 5

Parts List

Item	Price	Quantity
Mudd Pi Board + FPGA	~	1
Raspberry Pi 3.0	~	1
MCP601	\$0.34	1
AD623	\$6.72	1
ECG Pads	\$15 for 100	3
Alligator Clip Cables	Lab Stock	3
Wires + Passives	Stock	

Appendices

Appendix A: FPGA Block Diagram



Appendix B: Pi HTML code

```
//BBhr.html file
<!DOCTYPE html>

<html>

<head>

    <title> align = "center" BB’s EKG!</title>

    <meta http-equiv="content-type" content="text-html;charset=utf-8">

    <meta http-equiv="refresh" content="3">

</head>

<body>

    <h1 style="background-color:DodgerBlue;"><b> align = "center" BB’s EKG!</h1>

    <iframe  align = “center” src="cgi-bin/HRread" style="border:none;" height="500"
      width="500"></iframe>


</body>

</html>
```

Appendix C: Pi C code

```
// Brenden Brown, Benjamin Iten
// HRread.c
// Code for extracting HR

#include "SPI_GPIO.h"

char miso;
float miso2;

int main(void){
pioInit();
SPIInit();
spiInit(1250,0);
miso = spiSendReceive16(0x100);
miso2 = (float)miso;
printf("%s%c%c\n",
        "Content-Type:text/html;charset=iso-8859-1",13,10);
printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/cgi-bin/HRread.html \">");
printf("<html>\n");
printf("<body>\n");
printf("<h2>Your HR: ");
printf("%f", miso2);
printf("bpm</h2>\n");
printf("</body>\n");
printf("</html>\n");
return 0;
}
```

Appendix D: FPGA Verilog Modules

```
/*
Benjamin Iten  biten@g.hmc.edu
Brenden Brown bjbrown@g.hmc.edu
E155 Project: Electrocardiographic HR

This verilog code is used to take in an input from an external MCP3002 ADC, and
using SPI,
receive a ten bit reading. The FPGA is the
master and the ADC is the slave. The FPGA generates a chip select signal for
the ADC. It also
 serially communicates to the Pi an 8 bit heart rate for display on a webpage.

A divided serial clock is run at ~250Khz to communicate with the ADC,
and to the chip select signal is toggled every 16 clock cycles,
which means a new ADC reading is provided every 32 clock cycles. Then
another divided clock slows the rate at which samples enter data processing to
around 800hz.

Since the heart beat is so slow, a faster sampling rate is not needed.

Once an ADC reading is obtained it is passed to the signal processing along
with a corresponding
 done bit which controls a further divided clock to set the sampling rate of
the data processing
 at around 400 Hz. At the edge of this clock, a new data processing operation
begins on the most recent ADC reading.
A peak finding algorithm operates on finding the first sample that goes outside
of a dynamic
 envelope that is calculated based on the 200 most recent samples. Once the
number of samples
 between peaks is found, this is converted to the time domain and output as
beats per minute.
*/


module ECG(input logic clk,//40MHz
           input logic PiSCLK,//master clock from pi
           input logic reset,//from on board switch
           input logic mosi,
           input logic Din, //data from ADC, this is Dout on ADC pinout!
           output logic SCLK,//to ADC
           output logic CSBar,//to ADC
           output logic Dout,//to ADC   Din on schematic!!
           output logic miso);//to Pi
	logic done,newData;
	logic [9:0] ADC;//10-bit reading from ADC
	logic [7:0]HRout;//combined HR and voltage bits
	logic [9:0] sampleSlowClock;

	always_ff@(posedge done)
	begin //to slow down data processing rate to 100Hz
		sampleSlowClock <= sampleSlowClock+10'b1101000;//52 = 110100
	end

	assign newData = sampleSlowClock[9];

	SPIMaster spiM(Din,reset,clk,SCLK,Dout,CSBar,ADC,done);
	spiSlave spiS(PiSCLK,reset,miso,mosi,HRout);
	HRFind hrv(ADC,newData,reset,HRout);
endmodule

//module to communicate with MCP3002 ADC
//sends 250khz clock, and follows communication protocol from datasheet
module SPIMaster(input logic Din,//The MISO bit read each clock cycle from ADC
				input logic reset,
				input logic clk,
				output logic SCLK,
				output logic Dout, //MOSI bit to send to ADC
				output logic CSBar,
				output logic [9:0]Din10,//10 bit ADC num to be processed
				output logic done);//flag for when the entire ADC num is read

	logic [15:0] SCLKCounter; //Generates serial clock

	always_ff@(posedge clk)//Clock divider
		SCLKCounter<= SCLKCounter + 9'b110011010;//divided clk,250khz

	assign SCLK = SCLKCounter[15];

	logic [15:0]shiftReg; //inner sequence of bits to send to ADC to start
read cycle and read MISO bits
	logic [4:0]SPICycle; //to generate inner CSBar signal

	always_ff@(posedge SCLK)
		SPICycle <= SPICycle + 5'b1;//CSBar clock divider

	always_ff@(negedge SCLK)
	begin
		CSBar = SPICycle[4];//Chip select signal, goes high for 16 sclk
cycles at a time
		Dout = shiftReg[15]; //bit to send to slave
```

```
            if (~CSBar) shiftReg<={shiftReg[14:0],Din};//shift register to take
MISO bits into shift reg
            else if (SPICycle == 5'b11111)shiftReg<= 16'h6000;//mosi bit reset
      end

      always_ff@(posedge CSBar)
            Din10 <= shiftReg[9:0]; //sends ADC output to signal processing

      always_comb
            case(CSBar)
             1'b1: done = 1'b1;
             1'b0: done = 1'b0;
            endcase
endmodule

//module SPI Slave--adapted and inspired by DDCA 531.e15-16
//fpgaSEL asserted for one clock cycle then waits one clock cycle, then 8 bits
are sent to PI
//MSB is sent first, outputs zeros on tail end of signal

module spiSlave(input logic PiSCLK,
                input logic reset,
                output logic miso,
                input logic mosi,
                input logic [7:0]HRout );//combined signal of produced heart
rate and peak heartbeat);

      logic [7:0]HRsend;//inner module sample of the HRV input that can be
modified as a shift register

      always_ff@(negedge PiSCLK)
      begin
            if (mosi) HRsend= HRout;//on mosi bit, shift in HR, therefore Pi
only needs to send a single bit high
            else HRsend={HRsend[6:0],1'b0};
      end

      assign miso = HRsend[7];
endmodule

module HRFind(input logic [9:0]ADC,
              input logic newData,
              input logic reset,
              output logic[7:0] HRout);

      logic [17:0] movingAvg,baselineSum; //the average of the 200 most recent
samples, and the sum of last 200 samples
      logic [27:0] sampleHR;
      logic [9:0] subVal;//combined value from shift register to subtract from
200 value sum
      logic [14:0] count;//counter for samples between peakS
      logic [7:0] HR,baselineN;
      logic [15:0]peakSep;////total number of points between peaks
      logic [8:0]delta;//amount required for a sample to be outside of moving
envelope
      logic [199:0] bit0Reg, bit1Reg,  bit2Reg,  bit3Reg,  bit4Reg, bit5Reg,
bit6Reg,bit7Reg,bit8Reg,bit9Reg;
      logic startCount;
      logic [39:0]timeBetween;//time in microseconds between pulses


      always_ff@(posedge newData)
      begin
            if (reset)
            begin
                  {movingAvg}=18'b0;
                  count = 15'b0;
                  {bit0Reg,bit1Reg,bit2Reg,bit3Reg,bit4Reg,bit5Reg,bit6Reg,bit7
            Reg,bit8Reg,bit9Reg}=200'b0;
                  baselineN =8'b1;
                  baselineSum=18'h155;//initial guess that  baseline adc
reading is initially around 600
                  HRlast1<=8'b1000000;
                  sampleHR=8'b1000000;
                  HR = 8'b0; //reset HR to 0bpm
                  delta= 9'b10;//set value based on known circuit behavior
                  startCount = 1'b0;
            end
            else
            begin
                  movingAvg = baselineSum/baselineN;//moving average of the
baseline ADC readings
                  if (baselineN < 8'b11001000) baselineN <= baselineN+8'b1;
                  subVal ={bit9Reg[199], bit8Reg[199], bit7Reg[199],
bit6Reg[199], bit5Reg[199], bit4Reg[199], bit3Reg[199], bit2Reg[199],
bit1Reg[199], bit0Reg[199]};
                  baselineSum = baselineSum+ADC-subVal;//only adds values near
baseline to moving average
                  bit0Reg = {bit0Reg[198:0],ADC[0]};
                  bit1Reg = {bit1Reg[198:0],ADC[1]};
                  bit2Reg = {bit2Reg[198:0],ADC[2]};//shifts in most recent ADC
value,shifts out oldest
                  bit3Reg = {bit3Reg[198:0],ADC[3]};
                  bit4Reg = {bit4Reg[198:0],ADC[4]};
                  bit5Reg = {bit5Reg[198:0],ADC[5]};
                  bit6Reg = {bit6Reg[198:0],ADC[6]};
```

```
                    bit7Reg = {bit7Reg[198:0],ADC[7]};
                    bit8Reg = {bit8Reg[198:0],ADC[8]};
                    bit9Reg = {bit9Reg[198:0],ADC[9]};

                    if ((ADC > (movingAvg+delta))&&(startCount ==0))
                    begin
                          startCount = 1'b1;
                          count = 15'b1;
                    end

                    else if((count>15'b11100001)&&(ADC>(movingAvg + delta)))
                    begin
                          peakSep = count;
                          count = 15'b0;
                          startCount = 1'b0;
                          timeBetween = 16'b10011100010* peakSep; //converts from
num of samples to time in microseconds, 2500== 100111000100
                          sampleHR= (40'h3938700 / timeBetween); //=60 seconds in
microseconds divided by time between gets hr
                    end
                    else if (startCount)
                    begin
                          count = count+15'b1;
                    end
              end
      end
      assign HRout = sampleHR;
endmodule
```

Appendix E: Pi Library File

```
// SPI_GPIO.h

// Include statements
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

// GPIO FSEL Types
#define INPUT  0
#define OUTPUT 1
#define ALT0   4
#define ALT1   5
#define ALT2   6
#define ALT3   7
#define ALT4   3
#define ALT5   2

#define GPFSEL   ((volatile unsigned int *) (gpio + 0))
#define GPSET    ((volatile unsigned int *) (gpio + 7))
#define GPCLR    ((volatile unsigned int *) (gpio + 10))
#define GPLEV    ((volatile unsigned int *) (gpio + 13))
#define INPUT  0
#define OUTPUT 1

typedef struct
{
unsigned CS  :2;
unsigned CPHA :1;
unsigned CPOL :1;
unsigned CLEAR  :2;
unsigned CSPOL :1;
unsigned TA  :1;
unsigned DMAEN :1;
unsigned INTD  :1;
unsigned INTR  :1;
unsigned ADCS :1;
unsigned REN  :1;
unsigned LEN  :1;
unsigned LMONO  :1;
unsigned TE_EN :1;
unsigned DONE :1;
unsigned RXD :1;
unsigned TXD :1;
unsigned RXR  :1;
unsigned RXF  :1;
unsigned CSPOL0  :1;
unsigned CSPOL1  :1;
unsigned CSPOL2  :1;
unsigned DMA_LEN :1;
unsigned LEN_LONG :1;
unsigned  :6;
}spi0csbits;
#define SPI0CSbits (* (volatile spi0csbits*) (spi + 0))
#define SPI0CS (* (volatile unsigned int *) (spi + 0))

#define SPI0FIFO (* (volatile unsigned int *) (spi + 1))
#define SPI0CLK (* (volatile unsigned int *) (spi + 2))
#define SPI0DLEN (* (volatile unsigned int *) (spi + 3))

// Physical addresses
#define BCM2836_PERI_BASE        0x3F000000
#define GPIO_BASE               (BCM2836_PERI_BASE + 0x200000)
#define SYSTIMER_BASE           (BCM2836_PERI_BASE + 0x3000)
#define BLOCK_SIZE (4*1024)
#define SPI0_BASE               (BCM2836_PERI_BASE + 0x204000)

// Pointers that will be memory mapped
volatile unsigned int *gpio; //pointer to base of gpio
volatile unsigned int *spi; // pointer to base of systimer

void pioInit() {
int  mem_fd;
void *reg_map;

// /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
    printf("can't open /dev/mem \n");
    exit(-1);
}

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
    BLOCK_SIZE,       //Size of mapped memory block
    PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
    MAP_SHARED,       // This program does not have exclusive access to this memory
    mem_fd,           // Map to /dev/mem
    GPIO_BASE);       // Offset to GPIO peripheral

if (reg_map == MAP_FAILED) {
    printf("gpio mmap error %d\n", (int)reg_map);
    close(mem_fd);
```

```
      exit(-1);
    }

gpio = (volatile unsigned *)reg_map;
}

void SPIInit() {
int  mem_fd;
void *reg_map;

// /dev/mem is a psuedo-driver for accessing memory in the Linux filesystem
if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
      printf("can't open /dev/mem \n");
      exit(-1);
}

reg_map = mmap(
  NULL,             //Address at which to start local mapping (null means don't-care)
      BLOCK_SIZE,       //Size of mapped memory block
      PROT_READ|PROT_WRITE,// Enable both reading and writing to the mapped memory
      MAP_SHARED,       // This program does not have exclusive access to this memory
      mem_fd,           // Map to /dev/mem
      SPI0_BASE );

if (reg_map == MAP_FAILED) {
      printf("spi mmap error %d\n", (int)reg_map);
      close(mem_fd);
      exit(-1);
    }

spi = (volatile unsigned *)reg_map;
}

void pinMode(int pin, int function){
int reg = pin/10;
int offset = (pin%10)*3;
GPFSEL[reg] |= ((0b111&function)<<offset);
GPFSEL[reg] &= ~((0b111&~function)<<offset);
}


void spiInit(int freq, int settings) {
    //set GPIO 8 (CE), 9 (MISO), 10 (MOSI), 11 (SCLK) alt fxn 0 (SPI0)
    pinMode(8, ALT0);
    pinMode(9, ALT0);
    pinMode(10, ALT0);
    pinMode(11, ALT0);
    //Note: clock divisor will be rounded to the nearest power of 2
    SPI0CLK = 250000000/freq;   // set SPI clock to 250MHz / freq
    SPI0CS = settings;
    SPI0CSbits.TA = 1;          // turn SPI on with the "transfer active" bit
}

char spiSendReceive(char send){
    SPI0FIFO = send;            // send data to slave
    while(!SPI0CSbits.DONE);    // wait until SPI transmission complete
    return SPI0FIFO;            // return received data
}

short spiSendReceive16(short send) {
    short rec;
    SPI0CSbits.TA = 1;          // turn SPI on with the "transfer active" bit
    rec = spiSendReceive((send & 0xFF00) >> 8); // send data MSB first
    rec = (rec << 8) | spiSendReceive(send & 0xFF);
    SPI0CSbits.TA = 0;          // turn off SPI
    return rec;
}
```