Non-Invasive Heart Rate Monitor
E155 Final Project

Kristina Ming
Kaitlin Kimberling

December 9, 2014

1 Introduction

In this project, we create a system to measure a user's pulse. As the human heart pumps, a different volume of blood flows through a person's finger at a given time. This change in volume can be detected by shining light through the finger and then detecting the change in light reflected.

A PIC32 microcontroller is used for analog-to-digital conversion of the pulse wave as well as playing a sound corresponding to the user's pulse. An Altera Cyclone III FPGA is used for digital filtering of the signal as well as pulse determination. The PIC and FPGA communicate via the SPI protocol.

2 System Overview

The user's pulse is detected by measuring the reflectance of infrared light from an LED on an infrared photodiode, which produces a variable current output. This output is converted to a voltage output and amplifed before it is sent to the PIC32 to be converted to a digital signal. The digital signal is sent to the FPGA to be filtered. The FPGA also detects the peaks of the fitlered signal and counts them to produce a heart rate. The heart rate is displayed on a seven-segment display, and the pulse is sent to the PIC to be played on a speaker.

Figure 1 shows a general outline of the project and which subsystems use the PIC and FPGA.

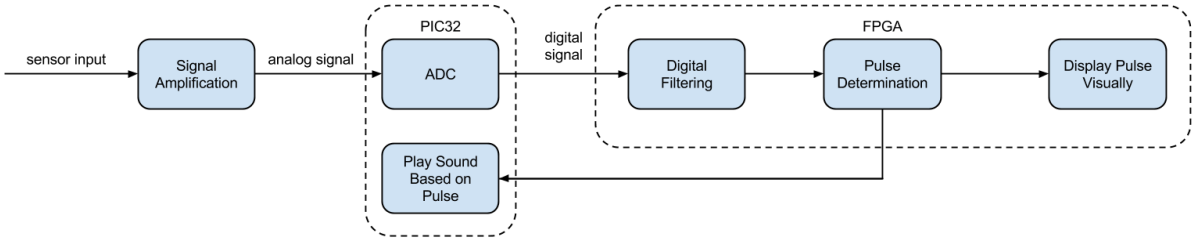

Figure 1: System Diagram

3 Analog Circuit Design

An infrared LED (with a wavelength of 850 nm) is used to send light through a user's finger. An infrared photodiode is then used to detect the change in reflectance of the light through the finger as the volume of blood changes. The photodiode acts as a current source that supplies varying current based on how much light it receives. The ADC on the PIC interprets variations in voltage levels, so we used a

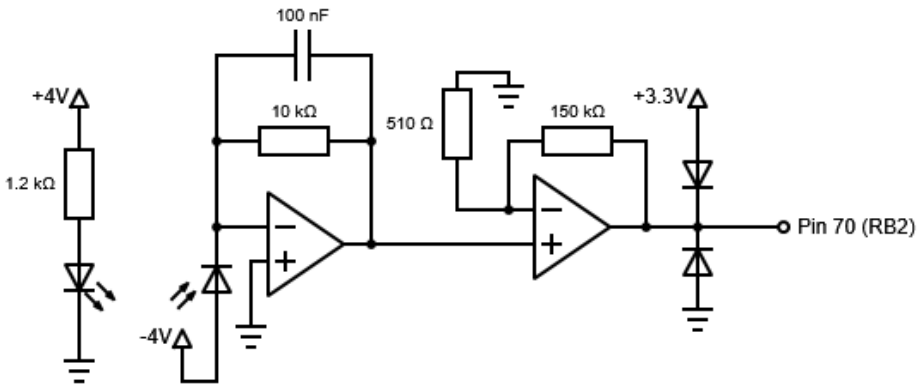

Figure 2: Diagram for sensor circuit

transimpedance amplifier to convert the current output to a voltage output.

It should be noted that the ADC on the PIC32 uses 0V and 3.3V as reference voltages. The values from the reflectance sensor were on the order of mV, so we added a non-inverting op-amp circuit to the output of the transimpedance amplifier to increase the range of the output signal. Figure 2 shows the diagram of the sensor and signal conditioning circuit. The 850 nm IR LED emits light and the photodiode detects the changes in light.

Additional analog circuitry needed to display the heart rate and play a sound are described in Appendix A. Appendix D lists the bill of materials for the project.

4 PIC32 Subsystem

The main purpose of the PIC is to convert the analog pulse signal into a digital signal and send it to the FPGA. It also receives the pulse from the FPGA and then plays the pulse on a speaker.

4.1 Analog-to-Digital Conversion

The PIC's built-in analog-to-digital converter (ADC) is used to convert the input signal from the amplification circuit into a digitized signal. The ADC is 10-bits and is left at the default reference voltages of 0-3.3V. Because heartbeat signal that we expect is low frequency (1-3 Hz), we chose to use a lower sampling rate to avoid oversampling and picking up too much high frequency noise. We thus configured the ADC to sample at 200 Hz. This is achieved by setting the peripheral clock divisor to 4 (to achieve a 40 MHz clock) and then setting timer 3 to a prescalar of 8. Waiting 6250 timer 3 cycles achieves the 200 Hz ADC sampling rate.

The sampled data is sent via 16-bit SPI communication. C code to interface with the ADC was adapted from the E155 DAC/ADC lecture[2].

We tested the ADC's functionality by writing the first 10,000 samples to a buffer, which we then exported and plotted in MATLAB. Figure 3 shows the a portion of these results. This plot indicates that the signal has high frequency noise which needs to be filtered before the peaks can be identified. The FPGA will take care of this filtering.

4.2 Playing the Heart Beat

After the ADC output is filtered on the FPGA and the peaks of the heartbeat are identified, the PIC causes a note to play on a speaker as each peak is identified, simulating the sound of an EKG machine.

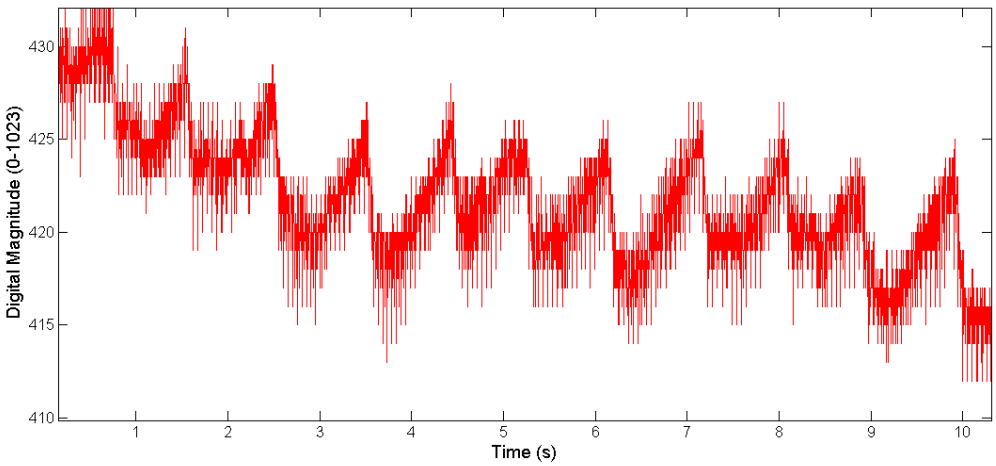

Figure 3: Raw ADC output

The speaker interface was taken from Lab 5, and the C code for playing notes was taken from the E155 lecture on C for GPIO[1]. The circuit for the speaker was taken from the LM386 audio amplifier data sheet[3].

5 FPGA Subsystem

The main function of the FPGA is to process the digital signal. The FPGA receives the digital voltage from the PIC via SPI and filters the signal using an FIR filter. It also detects the peaks of the signal and counts them to calculate a heart rate, which is displayed on a seven-segment display.

5.1 FIR Filter

The for the average human can range from around 60 BPM at rest to up to 200 BPM during physical activity. This translates to a signal of around 1-3 Hz. We thus need to filter out high frequency noise. We used MATLAB to design a 30th order lowpass filter with a cutoff frequency of 3.5 Hz. Figure 4 shows the frequency response of the FIR filter designed in MATLAB.

Figure 5 shows an example of unfiltered data from the ADC overlaid with data run through a simulation of the filter on MATLAB.
An Nth order FIR filter can be described by a difference equation in the following form, where the a_k's are coefficients determined by MATLAB.

$$y[n] = \sum_{k=0}^{N} a_k x[n-k]$$

In hardware, this can be described by a cascade of multiply-accumulate modules (MAC), as shown in Figure 6. Our 30th order filter is built with 31 coefficients. The coefficients are symmetric, so we only need 16 coefficients and therefore 16 MAC modules because a multiplier can be shared between two delayed signals with the same coefficient. The filter also has a shift register to store the delayed signals as they come in. Figure 7 shows how our filter is implemented in hardware.

We used TLC5620 serial DAC (digital-to-analog converter) to verify that the FPGA was filtering the signal properly. Figure 8 shows the raw, unfiltered data (bottom) and the data after being passed through the low-pass filter (top).

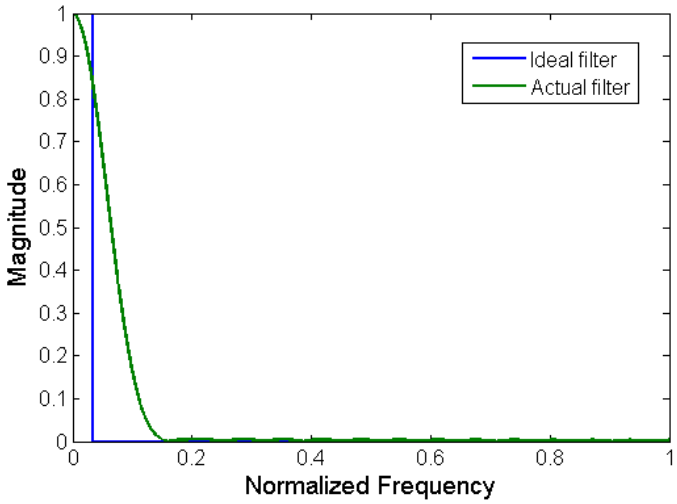

Figure 4: FIR filter designed in MATLAB, compared to an ideal filter

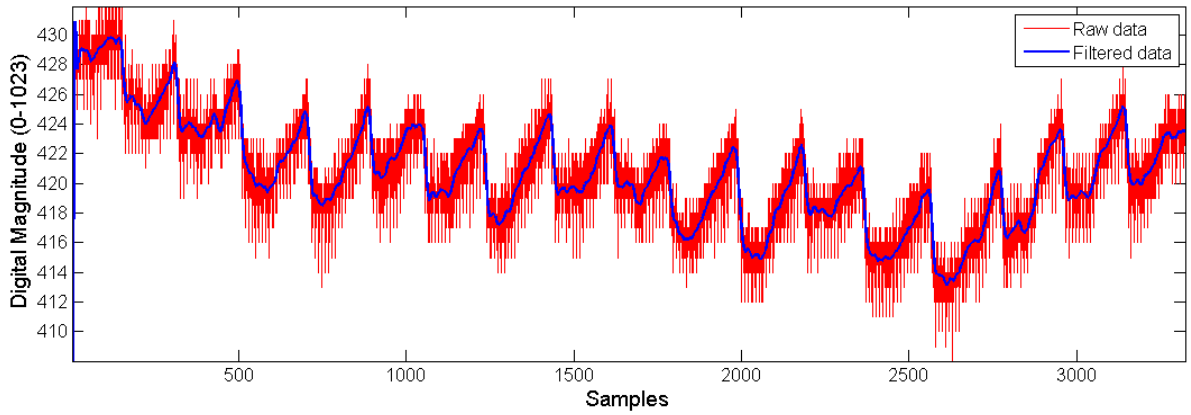

Figure 5: Filtered ADC Output

5.2 Peak Finding to Determine Heart Rate

Once we have a filtered signal, we need to identify the peaks which correspond to the pulse. There are several challenges to doing this. First, we want to identify the peaks in real time. Second, the signal tends to drift up and down, as seen in Figure 5. Third, even the filtered signal has small peaks that don't correspond to a pulse (such as the peaks in the trough), and we don't want to identify these as peaks. Many peak-finding algorithms define that a peak is found once it crosses a certain threshold value. This is an unreliable approach for our signal because the voltage level drifts up and down and varies from person to person.

We chose to use a rough numerical derivative approach to identify whether the signal is increasing or decreasing. A peak occurs when the signal transitions from mostly increasing to mostly decreasing. This is done by taking a buffer of 128 samples (sampled over 0.64 seconds) and subtracting each of the values from the buffer of values from the last time step. The differences are then converted to a 0 for a positive difference or a 1 for a negative difference. We can then split the buffer of 1's and 0's into two different arrays and add the 1's and 0's in each one. If the sum of the left-hand array is below a certain threshold (mostly increasing) and the sum of the right-hand array is above a certain threshold (mostly decreasing), then we have found a peak.

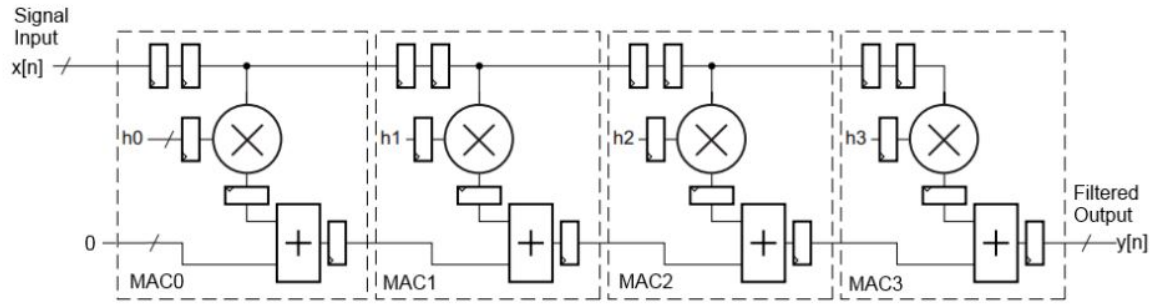

Figure 6: Cascade of multiply-accumulate modules[4]

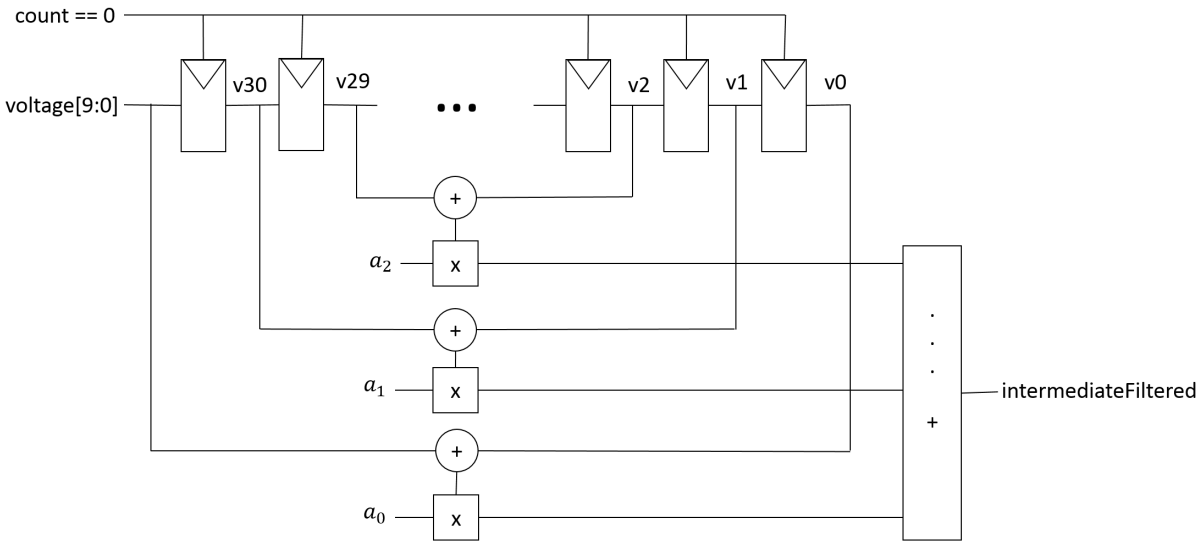

Figure 7: Overall diagram for FPGA filter implementation

A major downside to this setup is that it limits the speed of the pulse that can be detected. Because the buffer is 128 samples wide, a peak can only be detected at a maximum of every 0.64 seconds. This corresponds to a heart rate of 93.75 BPM. However, using a smaller buffer may cause false peaks to be detected.

5.3 Displaying the Heart Rate

The FPGA module that identifies peaks also counts the peaks that have been found. After the first three peaks (which are ignored to allow for noise from reset), a counter starts to count up to 10 seconds. The number of peaks detected during this time is multiplied by six to calculate the heart rate in beats per minute (bpm).

This heart rate is displayed on three seven-segment displays to allow for up to three digits of display. The input to two of the digits are multiplexed, and the third digit is controlled by separate pins. This is done because multiplexing three digits causes the display to be very dim. Figure 9 shows the layout of segments for the display. In SystemVerilog, segments A through G are seven13[0] and seven2[0] through seven13[6] and seven2[6], respectively.

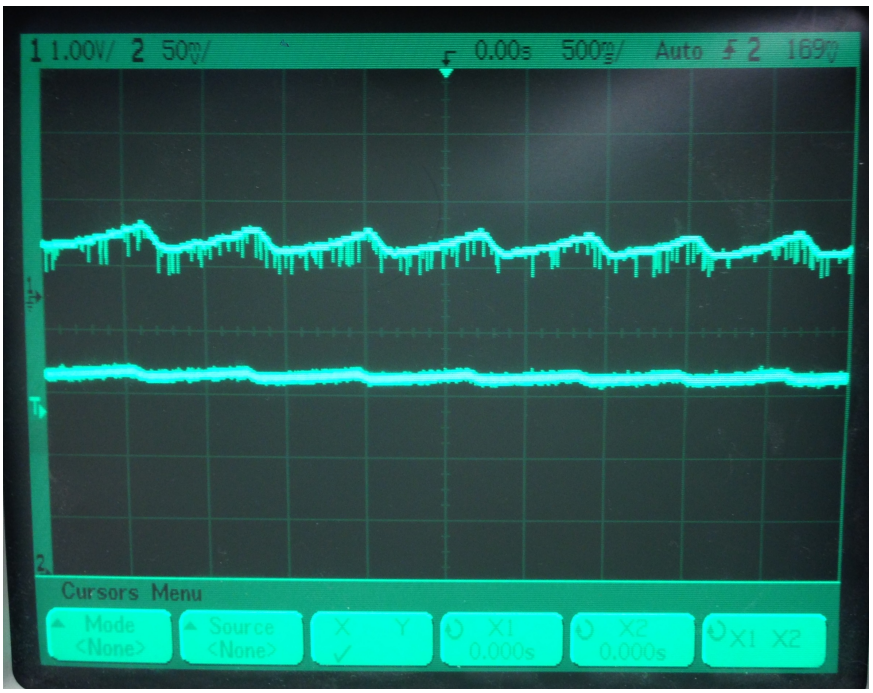

Figure 8: Original analog output (bottom) and the filtered signal from the DAC (top) on an oscilloscope

5.4 SPI Communication

The PIC and FPGA communicate via SPI. The baud rate was set to 1.25 MHz and the protocol was configured for 16-bit data transfer mode.

SPI is a serial communication protocol. In this system, the PIC acts as the master and the FPGA acts as the slave. Figure 10 shows an example of the SPI setup from the E155 lecture C for GPIO[1]. The difference is that our system uses a 16-bit shift register instead of an 8-bit shift register. Table 1 lists the internal pinout for SPI on the PIC.

Signal	**Pin**
clk	88
reset	71
sck	99
sdi	76
sdo	75

Table 1: Internal Pinout for SPI

6 Results and Discussion

Figure 11 shows the final setup. The system works as expected and displays a feasible heart rate for some users (approximately 54-66 beats per minute for a resting heart rate). The reliability of the system is dependent on many factors such as skin tone, skin thickness, and body mass. Each of these factors can make it more difficult to reflect light through a given user's finger and it is difficult to make a versatile system that works for all users.

Overall, the project is successful but in the future could be expanded to encompass more users. It is evident from some of the waves viewed on the oscilloscope that the photodiode detects light much better

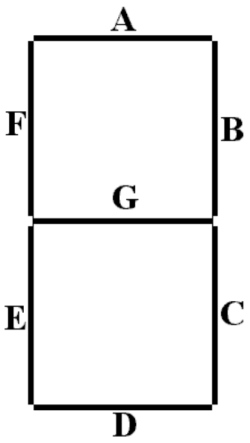

Figure 9: Seven Segment Display

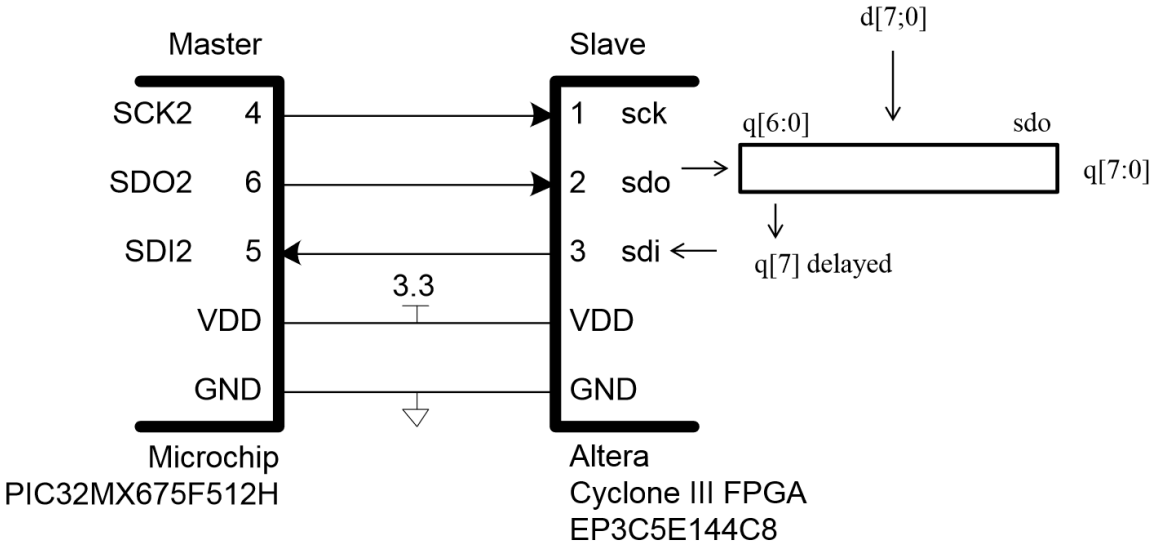

Figure 10: Example SPI Setup[1]

for some users. Consequently, a future system might not involve an IR LED and photodiode, but rather a different setup entirely to more reliably detect pulse.

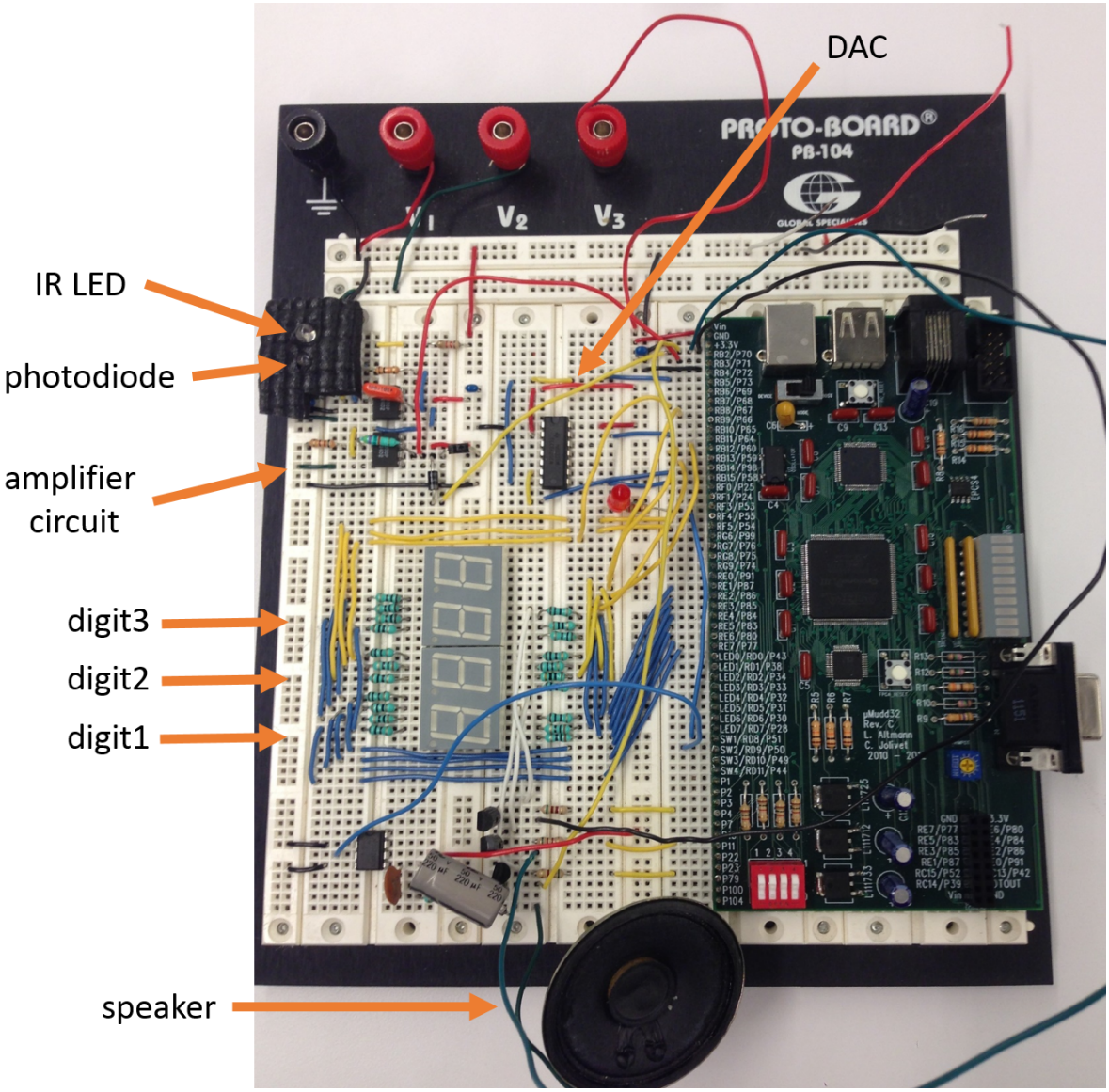

Figure 11: Final Setup

Appendix A: Circuit and System Diagrams

Figures 12, 13, and 14 show the diagrams for the DAC, seven-segment display, and speaker circuits, respectively.

Appendix B: FPGA SystemVerilog Code

```
// Kaitlin Kimberling and Kristina Ming
// E155 Final Project: Non-Invasive Heart Rate Monitor

/* signal processing code for FPGA */
module signal_processing(input logic clk, reset,
                         input  logic sck, sdo, sdi,
                         output logic peakLED, DACserial, load, LDAC, DACclk,
                         output logic [7:0] leds,
                         output logic disp1, disp2, disp3,
                         output logic [6:0] seven13, seven2);

        logic foundPeak;
        logic [9:0] filtered;
        logic [15:0] voltageOutput;
        logic [11:0] heartRate;
        logic [3:0] digit1, digit2, digit3;
        logic multiplex;
        logic [3:0] sevenIn;
        logic [7:0] numPeaks;
        logic [7:0] dummyLEDs;
        logic [28:0] count;

        // spi slave to get data from PIC
        spi_slave ss(sck, sdo, sdi, reset, d, q, voltageOutput);

        // filter to filter input signal from photodiode
        filter f1(reset, sck, voltageOutput[9:0], filtered);

        // find and count peaks of the filtered signal
        findPeaks128 peakFinder(clk, reset, sck, filtered[9:0],
    foundPeak, dummyLEDs, numPeaks, peakLED);
        countPeaks cp(clk, reset, foundPeak, heartRate, numPeaks, count);

        // output the filtered signal to the DAC for error checking
        DAC d1(sck, reset, filtered[9:0], DACserial, load, LDAC, DACclk);

        // multiplex display 1 and display 3
        multiplex2Displays chooseDisplay(clk, reset, multiplex, disp1, disp3);
        mux24 dispMux(digit1, digit3, multiplex, sevenIn);
        sevenSeg s13(sevenIn, seven13);
        sevenSeg s2(digit2, seven2);

        // display the heart rate in hex
        assign digit1 = heartRate[3:0];
        assign digit2 = heartRate[7:4];
        assign digit3 = heartRate[11:8];
        assign leds[7:0] = count[28:21];
endmodule

/* module to apply a digital FIR filter to an input signal */
module filter(input logic reset, sck,
                        input logic [9:0] voltage,
                        output logic [9:0] filteredSignal);
```

```
logic [3:0] count; // count to 32 (it takes 32 cycles to have all
                                    // of the SPI data

// filter coefficients
logic [31:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10;
logic [31:0] a11, a12, a13, a14, a15;

// delayed voltage values
logic [9:0] v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10;
logic [9:0] v11, v12, v13, v14, v15, v16, v17, v18, v19, v20;
logic [9:0] v21, v22, v23, v24, v25, v26, v27, v28, v29, v30;

logic [15:0] intermediateFiltered;

// 5-bit counter tracks when 32-bits is transmitted and new d should be sent
always_ff @(negedge sck, posedge reset)
        if (reset)
                count <= 0;
        else count <= count + 5'b1;

// assign FIR filter coefficients
always_comb
        begin
                // Multiplying by 1024 = 2^10
                a0 = 3;
                a1 = 4;
                a2 = 6;
                a3 = 8;
                a4 = 12;
                a5 = 17;
                a6 = 23;
                a7 = 29;
                a8 = 36;
                a9 = 43;
                a10 = 50;
                a11 = 56;
                a12 = 61;
                a13 = 65;
                a14 = 67;
                a15 = 68;
        end

// shift register to delay the voltage signal
always_ff @(posedge sck)
        if (count == 0)
                begin
                        v0 <= v1;
                        v1 <= v2;
                        v2 <= v3;
                        v3 <= v4;
                        v4 <= v5;
                        v5 <= v6;
                        v6 <= v7;
                        v7 <= v8;
```

```
                                        v8 <= v9;
                                        v9 <= v10;
                                        v10 <= v11;
                                        v11 <= v12;
                                        v12 <= v13;
                                        v13 <= v14;
                                        v14 <= v15;
                                        v15 <= v16;
                                        v16 <= v17;
                                        v17 <= v18;
                                        v18 <= v19;
                                        v19 <= v20;
                                        v20 <= v21;
                                        v21 <= v22;
                                        v22 <= v23;
                                        v23 <= v24;
                                        v24 <= v25;
                                        v25 <= v26;
                                        v26 <= v27;
                                        v27 <= v28;
                                        v28 <= v29;
                                        v29 <= v30;
                                        v30 <= voltage;

                                        // calculate the filtered signal
intermediateFiltered <= a0*(v0+v30) + a1*(v1+v29) + a2*(v2+v28) + a3*(v3+v27) +
a4*(v4+v26) + a5*(v5+v25) + a6*(v6+v24) + a7*(v7+v23) +
a8*(v8+v22) + a9*(v9+v21) + a10*(v10+v20) + a11*(v11+v19) +
a12*(v12+v18) + a13*(v13+v17) + a14*(v14+v16) + a15*v15;
                                        filteredSignal <= intermediateFiltered >> 10;
                                end
endmodule

/* SPI slave module */
module spi_slave(input logic sck, // from master
                                        input logic sdo, // from master
                                        output logic sdi, // to master
                                        input logic reset,
                                        input logic [15:0] d, // data to send
                                        output logic [15:0] q, // data received
                                        output logic [15:0] voltage); // discrete output

        logic [3:0] cnt;
        logic qdelayed;

        // 4-bit counter tracks when 16-bits is transmitted and new d should be sent
        always_ff @(negedge sck, posedge reset)
                if (reset)
                        cnt = 0;
                else cnt = cnt + 1;

        // loadable shift register
        // loads d at the start, shifts sdo into bottom position on subsequent step
        always_ff @(posedge sck)
                begin
```

```
                q <= (cnt == 0) ? d : {q[14:0], sdo};
                voltage <= (cnt == 0) ? q[14:0] : voltage;
        end

    // align sdi to falling edge of sck
    // load d at the start
    always_ff @(negedge sck)
        qdelayed = q[15];

    assign sdi = (cnt == 0) ? d[15] : qdelayed;

endmodule

/* module for DAC */
module DAC(input logic sck, reset,
           input logic [9:0] filteredSignal,
           output logic DACserial,
           output logic load, LDAC, DACclk);

    logic [1:0] A = 2'b00; // DAC channel for output
    logic RNG = 1'b0; // 1 times gain
    logic [10:0] buffer; // buffer to send output
    logic [3:0] count;
    logic [9:0] newSignal;

    always_comb
        begin
            LDAC = 1'b0;
            DACclk = sck;
        end

    // 4-bit counter tracks when 16-bits is transmitted and new d should be sent
    always_ff @(negedge sck, posedge reset)
        if (reset)
            begin
                count <= 1'b0;
            end
        else count <= count + 5'b1;

    always_ff @(posedge sck)
        if(count == 0)
            begin
                buffer <= {A,RNG,filteredSignal[7:0]};
                load <= 1'b1;
            end

        else if (count > 0 && count < 4'd12)
            begin
                DACserial <= buffer[10];
                buffer[10:0] <= {buffer[9:0], 1'b0};
            end

        // pull down load to send the data
        else if (count == 4'd12)
            load <= 1'b0;
```

```
                // pull load back up to transmit new data
                else if (count == 4'd13)
                        load <= 1'b1;
endmodule

/* module to find the peaks of a signal */
module findPeaks128(input  logic clk, reset, sck,
                                input  logic[9:0] newSample,
                                output logic foundPeak,
                                output logic [7:0] leftSumLEDS, numPeaks,
                                output logic newDiff);

        logic [3:0] sckcount;
        logic [9:0] oldSample, newDifference;
        logic [127:0] s; // shift register (buffer) to track slope change
        logic [9:0] leftSum, rightSum; // sum of left and right half of buffer
        logic [6:0] count; // 7-bit counter to keep track of how
    // long findPeak should stay high.

        // 4-bit counter tracks when 16-bits is transmitted and new d should be sent
        always_ff @(negedge sck, posedge reset)
                if (reset)
                        sckcount <= 0;
                else sckcount <= sckcount + 5'b1;

        // keep track of if the slope is increasing or decreasing
        always_ff @(posedge sck)

                if (reset)
                        begin
                                count <= '0;
                                leftSum <= 64;
                                rightSum <= 64;
                                foundPeak <= '0;
                                numPeaks <= 0;
                                s <= {128{1'b1}};
                        end

                else if (sckcount == 0)
                        begin
                                oldSample <= newSample;

                                // if the new value is greater than the old
                // value, the slope is increasing
                                // if the new value is less than the old value,
                // the slope is decreasing
                                // 0 for increasing slope, 1 for decreasing
                // slope
                                newDifference <= ~((newSample - oldSample) > 0);

                                // shift in the new indicator bit
                                s <= s << 1;
                                s[0] <= newDifference;
```

```
                        // keep track of the sum of the left and right
                // sides of the shift register
                        rightSum <= rightSum + newDifference - s[63];
                        leftSum <= leftSum + s[63] - s[127];

                        // LED output (for debugging)
                        leftSumLEDS[7:0] <= s[7:0];
                        newDiff <= foundPeak;

                        if ((leftSum <= 40) && (rightSum >= 38) && (count == 0)
                && (foundPeak == 0))
                                begin
                                        foundPeak <= 1'b1;
                                        numPeaks <= numPeaks + 1;
                                        count <= 7'b1;
                                end

                        // increment the counter if peak has been foundPeak
                        else if(foundPeak && (count != 0))
                                count <= count + 1'b1;

                        // wait time is over, turn off foundPeak
                        else
                                foundPeak <= 1'b0;
                end
endmodule


/* decoder for the seven segment display
   to display a single hexadecimal digit
   specified by an input s */
module sevenSeg(input  logic [3:0] s,
                output logic [6:0] seg);

    always_comb
        case(s)
            4'b0000: seg = 7'b100_0000; // 0
            4'b0001: seg = 7'b111_1001; // 1
            4'b0010: seg = 7'b010_0100; // 2
            4'b0011: seg = 7'b011_0000; // 3
            4'b0100: seg = 7'b001_1001; // 4
            4'b0101: seg = 7'b001_0010; // 5
            4'b0110: seg = 7'b000_0010; // 6
            4'b0111: seg = 7'b111_1000; // 7
            4'b1000: seg = 7'b000_0000; // 8
            4'b1001: seg = 7'b001_1000; // 9
            4'b1010: seg = 7'b000_1000; // A
            4'b1011: seg = 7'b000_0011; // B
            4'b1100: seg = 7'b010_0111; // C
            4'b1101: seg = 7'b010_0001; // D
            4'b1110: seg = 7'b000_0110; // E
            4'b1111: seg = 7'b000_1110; // F
            default: seg = 7'b000_0000;
        endcase
endmodule
```

```
/* module for a 2 input multiplexer (w/ 4-bit inputs) */
module mux24(input  logic [3:0] d0, d1,
                        input  logic s,
                        output logic [3:0] y);

        assign y = s ? d1 : d0;
endmodule

/* module to multiplex two seven segment displays based on
   a counter */
module multiplex2Displays(input  logic clk, reset,
                        output logic multiplex, disp1, disp2);

        logic [27:0] counter = '0;
        logic [27:0] thresh = 28'd250000;

        // the human eye can only see ~40 fps, so we toggle our display
        // at a rate above that
        always_ff @(posedge clk, posedge reset)
                if (reset)
                        begin
                                counter <= '0;
                                multiplex <= '0;
                        end

                else if (counter >= thresh)
                        begin
                                counter <='0;
                                multiplex <= ~multiplex;
                        end

                else
                        begin
                                multiplex <= multiplex;
                                counter <= counter + 1'b1;
                        end

        // choose which 7-segment display to use
        assign disp1 = multiplex;
        assign disp2 = ~multiplex;

endmodule

/* module to count the number of peaks over a certain time period
   and output the heart rate */
module countPeaks(input logic clk, reset, foundPeak,
                        output logic [11:0] heartRate,
                        input logic [7:0] numPeaks,
                        output logic [28:0] count);

        logic [28:0] thresh = 400000000; // Count up to 10s
        logic [3:0] periods = 6; // Multiply by this to get BPM

        always_ff @(posedge clk, posedge reset)
```

```
        begin
                if (reset)
                        begin
                                count <= '0;
                        end

                // increment the counter if we've counted at least two peaks
                // (the first two are erroneous measurements)
                else if(numPeaks > 2 && count < thresh)
                        begin
                                count <= count + 1'b1;
                        end

                // if we've reached the threshold, calculate the heart rate
                // and then stop
                else if (count == thresh)
                        begin
                                heartRate <= (numPeaks - 3) * periods;
                                count <= count + 1'b1;
                        end
        end
endmodule
```

Appendix C: PIC32 C Code

```
// Kaitlin Kimberling and Kristina Ming
// E155 Final Project: Non-Invasive Heart Rate Monitor

#include <P32xxxx.h>
#include <plib.h>

// function prototypes!
void initTimers(void);
void initspi(void);
int spi_send_receive(unsigned short send);
void initadc(int channel);
int readadc(void);
void playNote(unsigned short period, unsigned short duration);

// We want to sample at 200 Hz
// Divide clock by 4 (10 MHz clk) and have prescalar of 8

// initialize timers
void initTimers(void) {

        //      Assumes peripheral clock at 10MHz

        //      Use Timer3 for frequency generation
        //      T3CON
        //      bit 15: ON = 1: enable timer
        //      bit 14: FRZ = 0: keep running in exception mode
        //      bit 13: SIDL = 0: keep running in idle mode
        //      bit 12-8: unused
        //      bit 7:  TGATE = 0: disable gated accumulation
        //      bit 6-4: TCKPS = 011: 1:8 prescaler
        //      bit     3:      T32=0: 16-bit timer
        //      bit 2:  unused
        //      bit 1:  TCS = 0: use internal peripheral clock
        //      bit 0:  unused
        T3CON = 0b1000000000110000;

        // Timers for speaker

        //      Use Timer1 for note duration
        //      T1CON
        //      bit 15: ON = 1: enable timer
        //      bit 14: FRZ = 0: keep running in exception mode
        //      bit 13: SIDL = 0: keep running in idle mode
        //      bit 12: TWDIS = 1: ignore writes until current write completes
        //      bit 11: TWIP = 0: don't care in synchronous mode
        //      bit 10-8: unused
        //      bit 7:  TGATE = 0: disable gated accumulation
        //      bit 6:  unused
        //      bit 5-4: TCKPS = 11: 1:256 prescaler
        //      bit 3:  unused
        //      bit 2:  don't care in internal clock mode
        //      bit 1:  TCS = 0: use internal peripheral clock
        //      bit 0:  unused
```

```
        T1CON = 0b1001000000110000;

        //      Use Timer2 for frequency generation
        //      T2CON
        //      bit 15: ON = 1: enable timer
        //      bit 14: FRZ = 0: keep running in exception mode
        //      bit 13: SIDL = 0: keep running in idle mode
        //      bit 12-8: unused
        //      bit 7:  TGATE = 0: disable gated accumulation
        //      bit 6-4: TCKPS = 001: 1:2 prescaler
        //      bit     3:      T32 = 0: 16-bit timer
        //      bit 2:  unused
        //      bit 1:  TCS = 0: use internal peripheral clock
        //      bit 0:  unused
        T2CON = 0b1000000000010000;
}

// intialize SPI
void initspi(void) {
        int junk;

        SPI2CONbits.ON = 0; // disable SPI to reset any previous state
        junk = SPI2BUF; // read SPI buffer to clear the receive buffer
        SPI2BRG = 3; //set BAUD rate to 1.25MHz, with Pclk at 10MHz
        SPI2CONbits.MSTEN = 1; // enable master mode
        SPI2CONbits.CKE = 1; // set clock-to-data timing (data centered on
                        // rising SCK edge)
        SPI2CONbits.ON = 1; // turn SPI on
        SPI2CONbits.MODE16 = 1; // use 16-bit mode
}

// send and receive via SPI
int spi_send_receive(unsigned short send) {
        SPI2BUF = (send); // send data to slave
        while (!SPI2STATbits.SPIBUSY); // wait until received buffer fills,
                                        // indicating data received
        return SPI2BUF; // return received data and clear the read buffer full
}

// initialize ADC
void initadc(int channel) {
        AD1CHSbits.CH0SA = channel; // select which channel
        AD1PCFGCLR = 1 << channel; // configure input pin
        AD1CON1bits.ON = 1; // turn ADC on
        AD1CON1bits.SAMP = 1; // begin sampling
        AD1CON1bits.DONE = 0; // clear DONE flag
}

// read ADC
int readadc(void) {
        AD1CON1bits.SAMP = 0; // end sampling, start conversion
        while (!AD1CON1bits.DONE); // wait until DONE
        AD1CON1bits.SAMP = 1; // resume sampling
        AD1CON1bits.DONE = 0; // clear DONE flag
        return ADC1BUF0; // return result
```

```
}

// function to play a given note for a certain duration
void playNote(unsigned short period, unsigned short duration) {

        TMR1 = 0;       // Reset timers
        TMR2 = 0;

        while (TMR1 < duration) {       // Play until note ends
                if (period != 0) {              // Not a rest, so oscillate
                        PORTDbits.RD9 = 0;      // Output low
                        TMR2 = 0;
                        while (TMR2 < period) {}        // wait
                        PORTDbits.RD9 = 1;      // Output high
                        TMR2 = 0;
                        while (TMR2 < period) {}        // wait
                }
        }
}

int main(void) {
        TRISD = 0xF100;

        unsigned short ADCReadings[10000];
        int i = 0;

        // initialize timers and SPI
        initTimers();
        initspi();

        TMR3 = 0; // Reset timer
        int duration = 6250;
        unsigned short sample;
        unsigned short received;
        initadc(2); // use channel 2 (AN2 is RB2)

        while (1) {
                while(TMR3 < duration){
                        // wait
                }

                sample = readadc();
                PORTD = sample;
                TMR3 = 0; // reset timer

                if (i < 10000) {
                        ADCReadings[i] = sample;
                        i++;
                }

                // send data over SPI (offset the value so it will fall
                // within the 8 bits for the DAC)
                received = spi_send_receive(sample-300);

                if (PORTDbits.RD8 == 1) { // we received a pulse!
```

```
                playNote(527, 5);
            }
        }
}
```

Appendix D: Bill of Materials

Item	Quantity	Price
Capacitors	Various	N/A (Stockroom)
Resistors	Various	N/A (Stockroom)
Op007 Op Amp	2	N/A (Stockroom)
IR LED (850 nm)	1	$1
Photodiode	1	$2
TLC5620 Serial DAC	1	N/A (Stockroom)
LM386N Audio Amplifier	1	N/A (Stockroom)
Speaker	1	N/A (Stockroom)
Common Anode Seven-segment Display	2	N/A (Stockroom)

Table 2: Bill of Materials

References

[1] E155 C for GPIO Lecture. http://pages.hmc.edu/jspjut/class/f2014/e155/lectures/20141015_gpioc.pdf

[2] E155 DAC/ADC Lecture. http://pages.hmc.edu/jspjut/class/f2014/e155/lectures/20141103_dac.pdf

[3] LM386 Audio Amplifier Data Sheet. http://pages.hmc.edu/jspjut/class/f2014/e155/docs/LM386.pdf

[4] "Lab 3: Simulation and Testing". University of California, Berkeley Department of Electrical Engineering and Computer Sciences. http://www-inst.eecs.berkeley.edu/~cs150/fa13/lab3/lab3.pdf

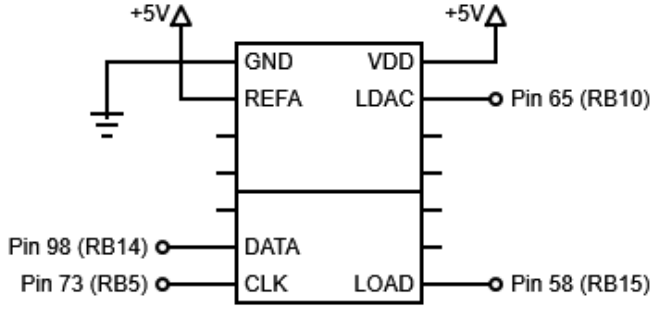

Figure 12: DAC circuit

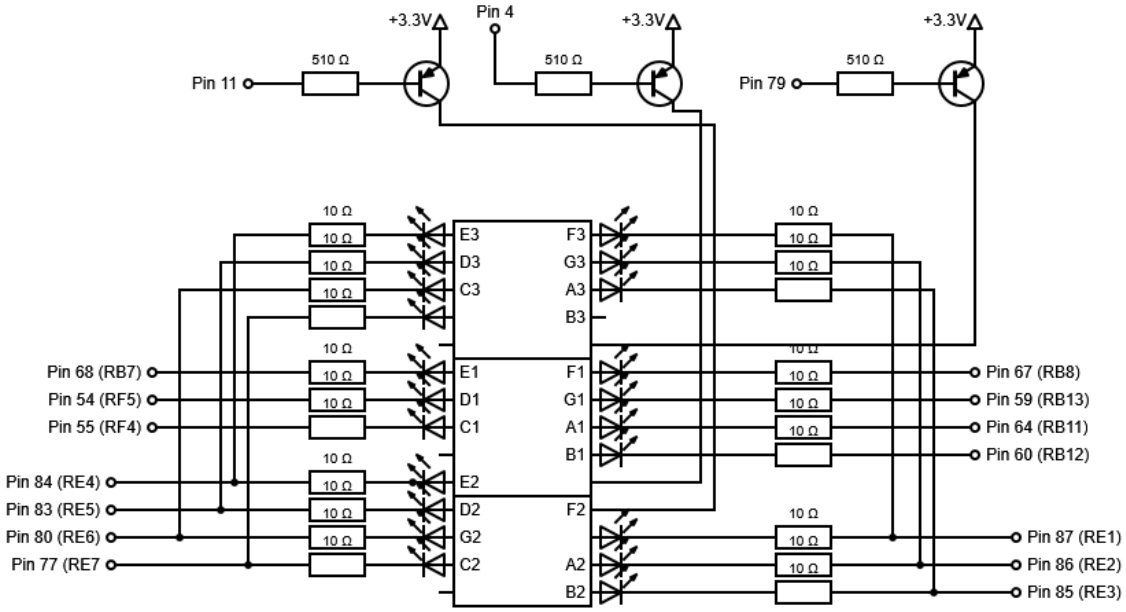

Figure 13: Seven-segment display circuit

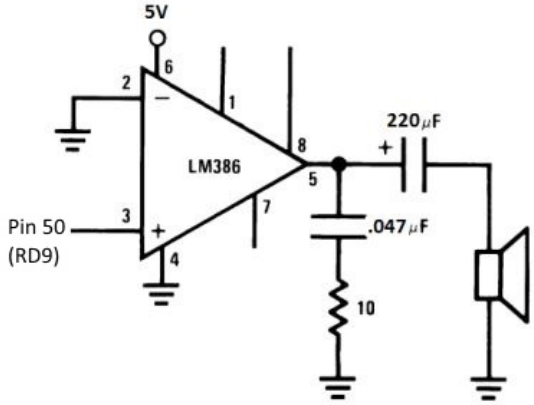

Figure 14: Speaker circuit