Heart Rate-Controlled Music Player

Final Project Report
December 9, 2016
E155

Senghor Joseph and Christine Goins

Abstract
This project implemented a heart rate-controlled music player to slow down or speed up music depending on the difference between the user's measured heart rate and their target heart rate. This can be used to influence a runner's pace to keep their heart rate in a safe or desired range. An FPGA uses the output of a pulse oximeter to determine the user's heart rate, which is sent to a Raspberry Pi, which uses the difference between the actual and target heart rates to control the speed of a song. The system measures the user's heart rate with sufficient accuracy and the Pi successfully receives the heart rate and changes the song speed when heart rate is significantly above or below the target heart rate.

Introduction

When running or doing other exercise, runners often try to get their heart rate into a certain range to achieve different goals such as warming up, fat-burning, or increasing endurance. Additionally, those recovering from heart surgery may need to keep their heart rate in a certain range during exercise to stay safe. Many runners already listen to music during their run, so music can be used to encourage runners to speed up or slow down their pace in order to affect their heart beat.

This project prototypes a music player whose speed is controlled by the difference between a user-input target heart rate and the user's actual measured heart rate. The user inputs their target heart rate, the heart rate they would like to reach during their activity, through DIP switches on the breadboard. The user clips the pulse sensor onto their ear and the FPGA calculates the user's heart rate from output of the sensor, which is put through an analog-to-digital converter to be readable by the FPGA. This calculated heart rate is always displayed on the LEDs on the μMudd Mark IV board. When the user begins the music player by clicking a button on the Raspberry Pi-hosted website, the Pi begins requesting and receiving the heart rate from the FPGA using SPI. The Pi compares this heart rate to the target heart rate, and computes a speed factor based on the difference between the two heart rates. This speed factor controls the speed of the music played from the Pi using an audio power amplifier and a speaker. Figure 1 shows a block diagram of the system.

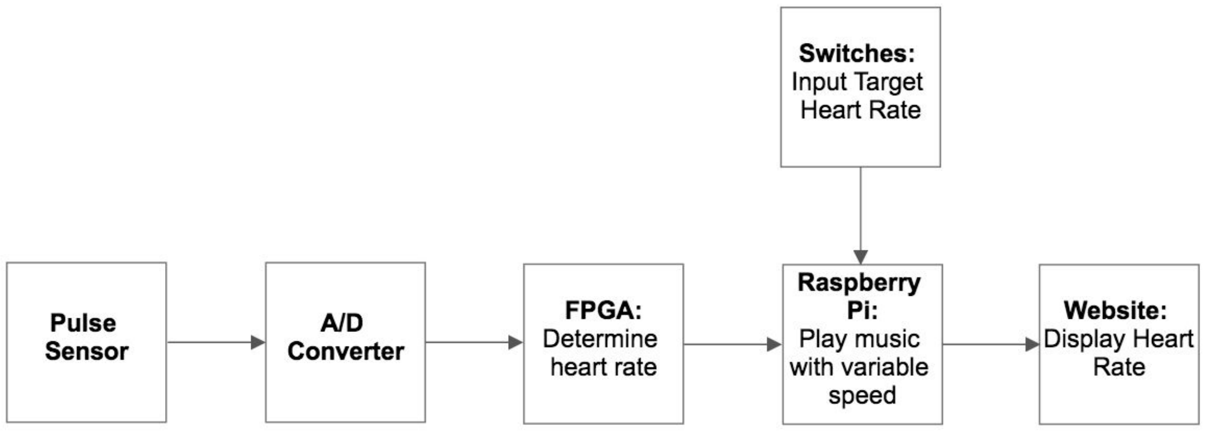

Switches: Input Target Heart Rate
Pulse Sensor
A/D Converter
FPGA: Determine heart rate
Raspberry Pi: Play music with variable speed
Website: Display Heart Rate

Figure 1. Block diagram of system

New Hardware

The project uses a pulse oximeter that, when pressed firmly against an earlobe or finger, measures differences in light absorption in a user's skin. It uses this to output a voltage corresponding to the amount of oxygenated hemoglobin in the blood, which increases with each heartbeat and decreases as the body uses it up. The sensor is simple to use, with only three wires: power (3.3 or 5V), ground, and output. When using the sensor the designer should note that the output varies highly depending on the user. For some users the output is so large that it is clipped, yielding a flat line at the top of the signal rather than a peak. For other users the output is so small that noise is almost as large as the desired heartbeat signal. These differences are due to differences in skin tone and where blood vessels lie in an individual's earlobe.

Schematics

The schematic of the entire system is shown below in Figure 2.

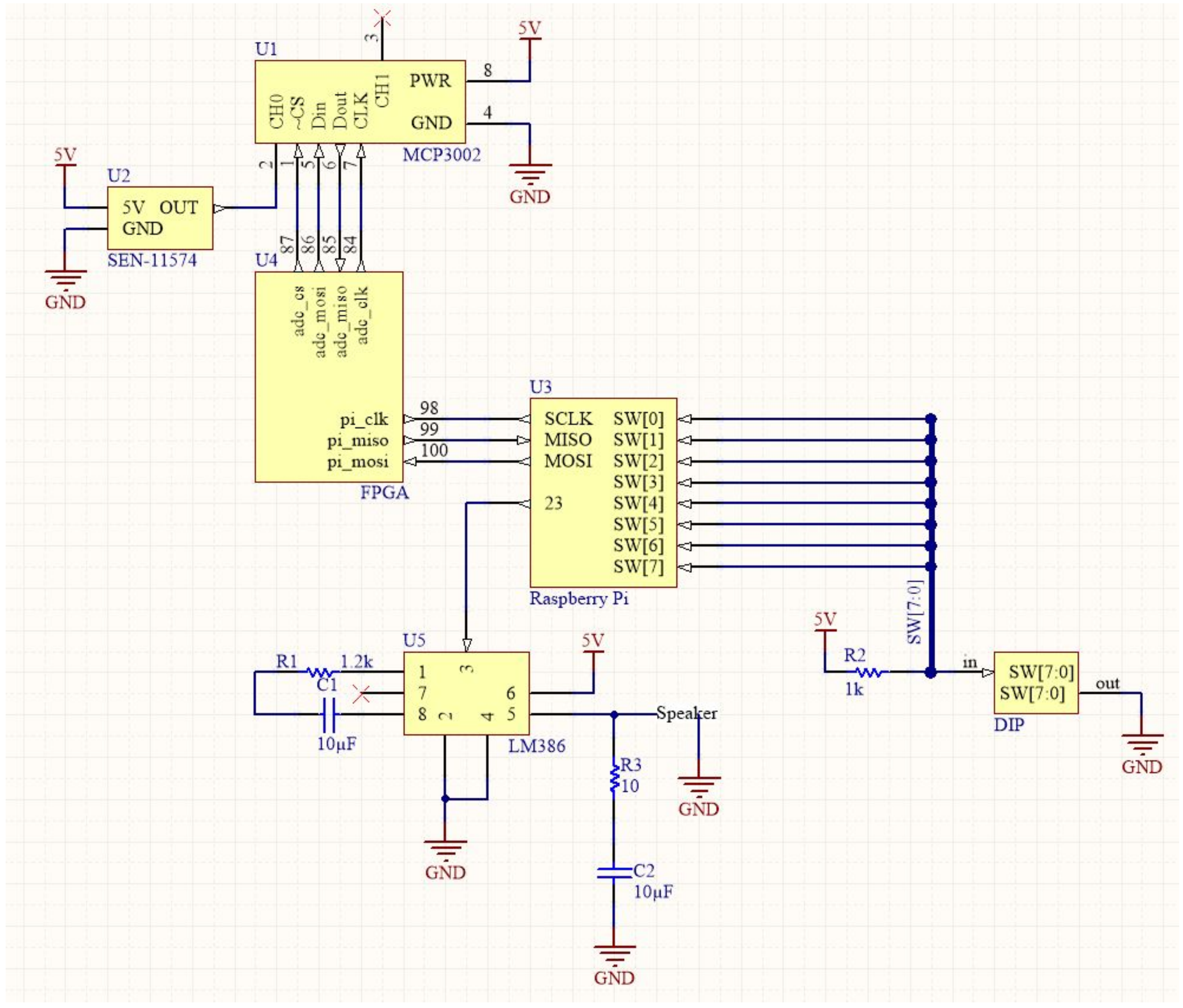


Figure 2. Circuit schematic

The PPG (photoplethysmogram), referenced in the schematic as the *SEN-11574*, has an output that correlates very well with the amount of light oxygen in the blood absorbs or reflects. This generates an analog signal similar to a heartbeat waveform.

The analog-to-digital converter (ADC), called the MCP3002 in the schematic, then takes that analog signal and converts it into a digital signal that the receiving FPGA can process and transfers this information with a serial protocol called SPI.

The FPGA takes the digital signal from the ADC over SPI and peak detects the signal in order to calculate the heart rate. It then stores this heart rate and waits until the Raspberry Pi requests it.

The Raspberry Pi requests a recent heart rate from the FPGA and compares this value to a target heart rate the user has defined. Depending on whether the requested heart rate is higher or lower than the target, the playback speed of the music the Pi is outputting into an audio amplifier will decrease or increase.

The LM386 is an audio amplifier that increases the gain of its input signal by 50 (as it has been set up in the circuit) and outputs the signal into an 8 ohm speaker.

FPGA Design

The FPGA calculates the user's heart rate. It takes in the output of the pulse sensor, uses thresholding to detect each pulse, computes the heart rate, and sends it to the Raspberry Pi using SPI. The flow of data is shown in Figure 3.

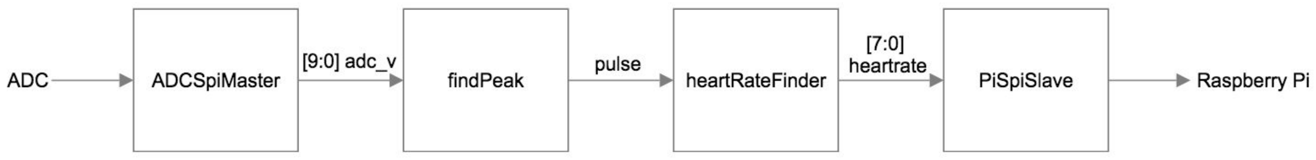


Figure 3. FPGA System Diagram

Because the output of the pulse sensor is an analog signal, it cannot be directly input to the FPGA. The sensor output if first put through a 10-bit analog-to-digital converter (ADC) (MCP3002), which the FPGA reads using the module SPIMasterADC. The 10-bit output of the ADC is sent serially to the FPGA using the timing diagram in the MCP3002's datasheet. The FPGA controls the clock speed of the ADC, sets CS low to begin data transfer, and sends the ADC the required bits to receive data in the desired format. It also uses a shift register to shift in the data it receives from the ADC.

Once all 10 bits from the ADC (*adc_v*) have been received by the FPGA, the thresholding module findPeak stores *adc_v* and checks whether it is greater than the threshold of 600 which corresponds to a voltage of 2.93 V. Because the desired output of this module is a single pulse when the signal crosses the threshold, a pulse is only generated if the previous input to findPeak was not greater than the threshold.

HeartRateFinder receives the pulses generated by findPeak and uses the time between them to determine the heart rate. The heart rate is equal to (# clk cycles per second * 60 sec/min)/(# clk cycles between pulses) = 2,400,000,000/(# clk cycles between pulses). A counter counts the number of clock cycles between pulses and restarts the count when an acceptable pulse is received. Because this module uses the previous heart rate to determine whether the next heart rate is plausible, it does not accept the first calculated heart rate. Instead it compares the average of the first three heart rates with the fourth, and only accepts the fourth heart rate if it is within ±25% of the average. Heart rates less than 30 BPM or greater than 250 BPM are not allowed because physically these heart rates are very unlikely. Additionally a heart rate that is more than double or less than half of the previous heart rate is not allowed because this extreme acceleration or deceleration is very unlikely. The FSM for this process is included in Appendix B.

The calculated heart rate from heartRateFinder is displayed in binary on the LEDs of the µMudd Mark IV board, and is also sent to the Raspberry Pi using SPI when the Pi requests it.

Raspberry Pi Design

The Raspberry Pi receives the user's heart rate from the FPGA and the target heart rate from the switches, then plays a song at a speed dependent on the difference between the two heart rates by outputting a square wave for various frequencies and durations.

When the program is run, the Pi begins a loop in which each iteration plays one note of the song Fur Elise. In each iteration the Pi requests and reads a heart rate from the FPGA using SPI, while also keeping track of past heart rates and calculating an average. Every tenth iteration, the program reads a heart rate from the FPGA, then compares the average of the past 10 heart rates to the target heart rate. The difference between the two is used to calculate the speed factor, which is used to speed up or slow down the music. If the user's heart rate is above the target, then the music slows down, and if it is below, it speeds up. The amount by which it speeds up or slows down is dependent on the number of BPM that the actual and target heart rate differ by. The program continues until all notes of the song have been played.

To request and receive heart rates from the FPGA, the program first sets the clock frequency to 122 kHz, the phase and polarity to 0, and the TA bit to 1 to enable SPI. Each time the programs asks for a new heart rate, it uses SPIsendReceive to send eight 0s to the FPGA. Because SPI is a built-in peripheral of the Pi, there is no need to explicitly generate a clock for the data transfer.

The target heart rate is input in binary using eight switches. The program sets the pins connected to the switches to inputs and performs the proper logic in order to use the target heart rate to calculate the speed factor.

The program uses an array of notes, where each note specifies a frequency and duration, to play the song. A square wave is generated by using timers to alternately write 1 and 0 to the outpin for a length of half the note's period. Another timer is used to continue this process for the given duration. The duration is determined by dividing the note's typical duration by the calculated speed factor.

The Pi also hosts a website using an Apache2 web server. The website prints the user's averaged heart rate using an iframe, which is an inline frame used to embed another HTML page within the current page, allowing the heart rate to update without leaving the web page.

Results

The majority of the time the system successfully calculates the user's heart rate, but occasionally the heart rate vastly increases by almost double, likely due to a false peak being detected. However with the averaging in the Raspberry Pi code, the heart rate that determines the speed of the music accurately reflected the actual heart rate. The team implemented the Pan Tompkins algorithm for pulse detection, and while its outputs in ModelSim matched those calculated in MATLAB, when tested in hardware, thresholding with a constant value performed better than the algorithm. However to make this system perform more successfully on different users, for whom which the pulse sensor output can vary greatly in both magnitude and shape, an algorithm such as Pan Tompkins would be more successful than constant thresholding.

The team planned to have the user input the target heart rate through the website, but could not successfully implement this without making the program restart when a new target heart rate was entered, so instead used physical switches connected to input pins of the Pi, so that the user could enter the input in binary.

The website successfully printed the user's average measured heart rate and the input target heart rate.

The music had jumps between and during notes, which could be improved by generating sine waves using a D/A converter or using the built-in audio output of the Pi. This would require a different method of changing the speed of the music.

References

[1] Pulse Sensor http://pulsesensor.com/
[2] Hardware Implementation of Pan & Tompkins QRS Detection Algorithm: http://mule.cslab.ece.ntua.gr/docs/c8.pdf

Parts List

Part	Source	Vendor Part #	Price
Pulse Sensor Amped	SparkFun	SEN-11574	$24.95

LM386 Low Voltage Audio Power Amplifier	E155 cabinet	LM386	N/A
Speaker	Desktop Computer	None	N/A
DIP Switch	E155 cabinet	N/A	N/A

Appendix A : SystemVerilog Code

```
////////////////////////////////////////////////////////////////////////////////////
/
/* E155 Final Project: Heart Rate-Controlled Music Player
       Senghor Joseph and Christine Goins
       sjoseph@hmc.edu    cgoins@hmc.edu
       This project calculates a heart rate based on a 10-bit output from an ADC
connected
       to a pulse oximeter and sends heart rate to a Raspberry Pi using SPIMasterADC
*/
////////////////////////////////////////////////////////////////////////////////////
/
module heartRateCalc(input logic pi_clk, clk, reset,
                     input logic pi_mosi, adc_miso,
                     output logic pi_miso, adc_mosi, adc_clk, adc_cs,
                     output logic [7:0] led,
                     output logic pulse);

       logic ext_clk, newData;
       logic [12:0] clkcount;
       logic [9:0] adc_v;
       logic [7:0] hr;
       logic [7:0] pi_data; // never used

       // 6.4 kHz clk generator
       always_ff@(posedge clk)
              if(reset)
              begin
                     clkcount <= 0;
                     ext_clk <= 0;
                     end
              else if(clkcount == 13'd3125)
                     begin
                     ext_clk <= ~ ext_clk;
                     clkcount <= 0;
                     end
              else
                     begin
                     clkcount <= clkcount + 1'b1;
                     end

       spiSlavePi pi_spi(pi_clk, pi_mosi, hr, pi_miso, pi_data);

       spiMasterADC adc_spi(ext_clk, reset, adc_miso, adc_mosi, adc_cs, newData,
adc_clk, adc_v);

       findPeak findPeak(clk, reset, newData, adc_v, pulse);

       heartRateFinder getHR(clk, reset, pulse, hr);
```

```
	// display heart rate in binary on board LEDs
	assign led = hr;

endmodule


////////////////////////////////////////////////////////////////////////////////////
/
/* findPeak
	E155 Final Project: Heart Rate-Controlled Music Player
	Senghor Joseph and Christine Goins
	sjoseph@hmc.edu    cgoins@hmc.edu
	This module compares the outpt of the ADC with a constant threshold to obtain
	a pulse on the rising edge the output exceeds the threshold
*/
////////////////////////////////////////////////////////////////////////////////////
/
module findPeak(input logic clk, reset, newData,
		input logic [9:0] x_in,
		output logic pulse);

	logic [9:0] x0, threshold;
	logic oldy_out, y_out;
	assign threshold = 10'd600;

	// store new value everytime there is a new ADC output
	flopenr x0f(clk, reset, newData, x_in, x0);

	// does ADC output exceed threshold?
	assign y_out = (x0 > threshold);

	flopr #(1) yflop(clk, reset, y_out, oldy_out);

	// to make pulse only go high on posedge of y_out
	always_ff@(posedge clk, posedge reset)
		if(reset) pulse <=0;
		else if(y_out & ~oldy_out) pulse <= 1;
		else pulse <=0;

endmodule


////////////////////////////////////////////////////////////////////////////////////
/
/* heartRateFinder
	E155 Final Project: Heart Rate-Controlled Music Player
	Senghor Joseph and Christine Goins
	sjoseph@hmc.edu    cgoins@hmc.edu
	This module takes the pulses from findPeak and determines which are acceptable
	to calculate the heart rate
```

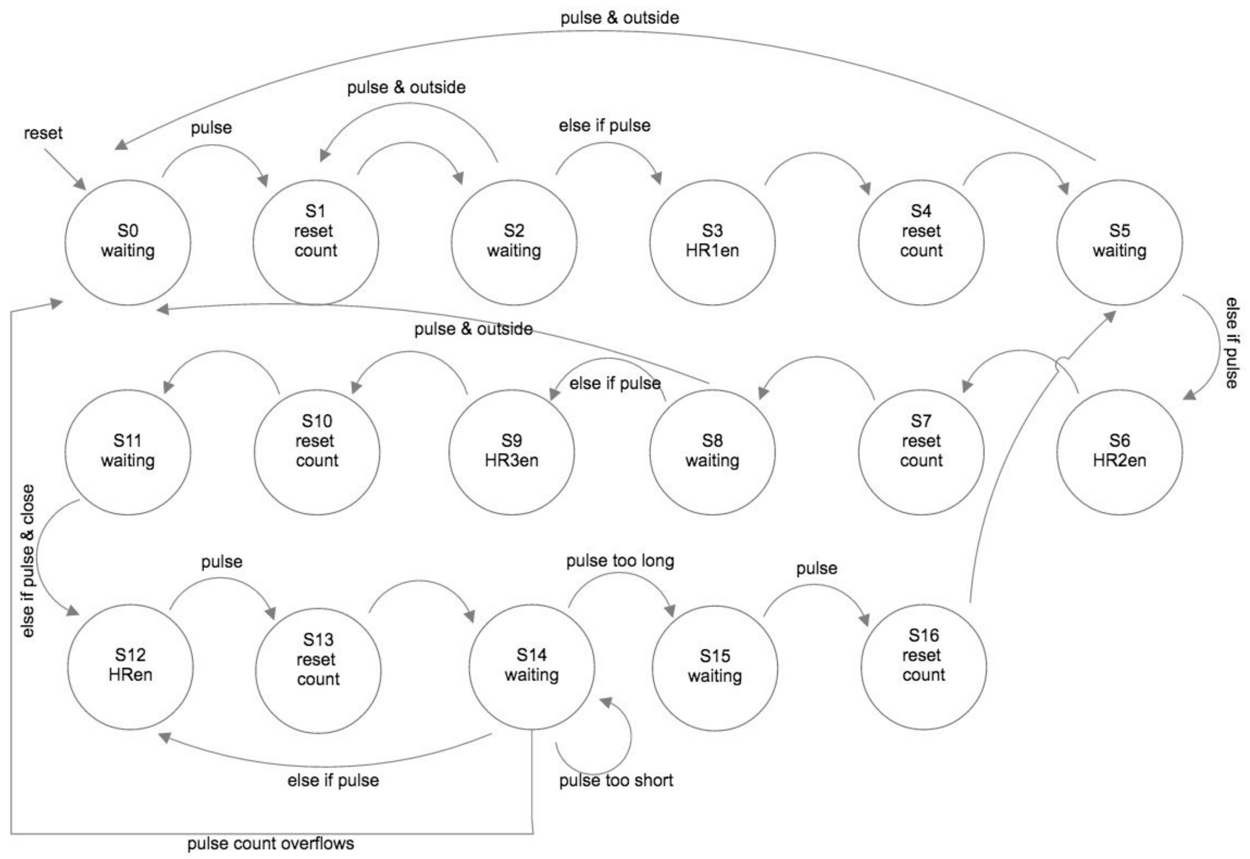
```
*/
/////////////////////////////////////////////////////////////////////////////////
/
module heartRateFinder(input logic clk, reset, pulse,
                       output logic [7:0] HR);

       logic counterReset, countReset, HRen, HR1en, HR2en, HR3en, close;
       logic [27:0] pulseCount;
       logic [7:0] interHR, HR1, HR2, HR3, avg;

       // counts # of clks between pulses
       counter #(28) pulseCounter(clk, counterReset, pulseCount);
       assign counterReset = reset | countReset;

       // calculates what the HR would be at the current count
       assign interHR = (32'd2400000000)/pulseCount;

       // new HR is stored when flop is enabled by HRen
       flopenr #(8) interHRflop(clk, reset, HRen, interHR, HR);

       // store first 3 acceptable heart rates
       flopenr #(8) HR1flop(clk, reset, HR1en, interHR, HR1);
       flopenr #(8) HR2flop(clk, reset, HR2en, interHR, HR2);
       flopenr #(8) HR3flop(clk, reset, HR3en, interHR, HR3);

       // FSM to determine heart rate
       typedef enum logic [4:0] {s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,
s12, s13, s14, s15, s16} statetype;
       statetype state, nextstate;

       always_ff@(posedge clk, posedge reset)
              if(reset) state <= s0;
              else state <= nextstate;

       always_comb
              case(state)
                     s0: if(pulse) nextstate = s1; // begin waiting
                            else nextstate = s0;
                     s1: nextstate = s2; // reset count
                     s2:  if(pulse & outsideRange) nextstate = s0; // reset FSM
                            else if (pulse) nextstate = s3;
                            else nextstate = s2; // wait for pulse
                     s3: nextstate = s4; // HR1enable
                     s4: nextstate = s5; // reset count
                     s5:  if(pulse & outsideRange) nextstate = s0; // reset FSM
                            else if (pulse) nextstate = s6;
                            else nextstate = s5; // wait for pulse
                     s6: nextstate = s7; // HR2enable
                     s7: nextstate = s8; // reset count
                     s8:  if(pulse & outsideRange) nextstate = s0; // reset FSM
```

```
                                else if (pulse) nextstate = s9;
                                else nextstate = s8; // wait for pulse
                        s9: nextstate = s10; // HR3enable
                        s10: nextstate = s11; // reset count
                        s11:  if(pulse & outsideRange) nextstate = s0; // reset FSM
                                else if (pulse & close) nextstate = s12; // if HR is close
to average of previous 3 HRs
                                else nextstate = s11; // wait for pulse
                        s12: nextstate = s13; // HR enable
                        s13: nextstate = s14; // reset count
                        s14: if(pulse & (interHR > HR<<1 | interHR > 8'd250)) nextstate =
s15; // if HR is too big
                                else if (pulse & (interHR < HR>>1'd1 | interHR < 5'd30))
nextstate = s14; // if HR is too small, keep waiting and counting
                                else if (pulse) nextstate = s12;
                                else if (pulseCount == 28'd268435455) nextstate = s0; //
if counter overflows, start over
                                else nextstate = s14;
                        s15: if(pulse) nextstate = s16;
                                else nextstate = s15;
                        s16: nextstate = s5; // reset count and maintain previous HR
                endcase

        // state-dependent output logic
        assign countReset = (state==s1) | (state==s4) | (state==s7) | (state==s10) |
(state==s13) | (state==s16);
        assign outsideRange = (interHR < 5'd30 | interHR > 8'd250); // if HR is too big
or small
        assign HRen = (state==s12);
        assign HR1en = (state==s3);
        assign HR2en = (state==s6);
        assign HR3en = (state==s9);
        assign avg = ((HR1 + HR2 + HR3)/3); // average of first 3 heart rates
        assign close = ((avg - (avg>>2)) < interHR) & (interHR < (avg + (avg>>2))); //
true if interHR is within +/-25% of avg

endmodule


/////////////////////////////////////////////////////////////////////////////////////
/
/* spiMasterADC
        E155 Final Project: Heart Rate-Controlled Music Player
        Senghor Joseph and Christine Goins
        sjoseph@hmc.edu    cgoins@hmc.edu
        This module generates a clock, CS, and the proper output to receive the ADC's
10-bit
        output using a shift register. The clock runs at 3.2kHz for a sampling rate of
200Hz
*/
```

```
////////////////////////////////////////////////////////////////////////////////////
/
module spiMasterADC(input logic ext_clk, reset, miso,
                    output logic mosi, CS, miso_en, adc_clk,
                   output logic [9:0] misoData);

       // use counter to generate correct outputs and enables
       logic [4:0] count;
       counter #(5) spiCount(ext_clk, reset, count);

       // shift register to get misoData
       always_ff@(posedge clk, posedge reset)
              if(reset) misoData <= 10'b0;
              else if(den) misoData <= {misoData[9:1], miso};

       assign CS = (count==5'd30) | (count==5'd31); // CS goes low at beginning of
data transfer
       assign adc_clk = ((count%2)==1); // generate 3.2kHz clk (adc_clk is 1 when clk
count is odd)
       assign mosi = (count==5'd0) | (count==5'd1) | (count==5'd6) | (count==5'd7); //
fPGA sends 8'b00110100
       assign den = (count>5'd10) & (count<5'd30) & ((count%2)==1); // enabling shift
register (enable on odd counts between 10 and 30)
       assign miso_en = (count==5'd0); // when all 10 bits have been receieved

endmodule


////////////////////////////////////////////////////////////////////////////////////
/
/* spiSlavePi
       E155 Final Project: Heart Rate-Controlled Music Player
       Senghor Joseph and Christine Goins
       sjoseph@hmc.edu    cgoins@hmc.edu
       This module takes in the Raspberry's Pi's clk and output so that it can send
the heart
       rate to the Pi at the correct time
*/
////////////////////////////////////////////////////////////////////////////////////
/
module spiSlavePi(input logic sck, // from master
                  input logic mosi, // from master
                  input logic [7:0] d, // data to send
                  output logic miso, // to master
                  output logic [7:0] q);  // data received

              logic [2:0] count;
              logic qdelayed;

              // counter to track when full byte is transmitted
```

```
             always_ff@(negedge sck)
                    count = count + 3'b1;

             // shift register that starts with d and shifts mosi in
             // at each clk so that q ends up with 8 bits from master
             always_ff@(posedge sck)
                    q <= (count == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

             // miso aligned w/ falling edge of sck
             always_ff@(negedge sck)
                    qdelayed = q[7];
             assign miso = (count == 0) ? d[7] : qdelayed;

endmodule

// parametrized Enabled DFF
module flopenr #(parameter WIDTH = 10)
             (input logic clk, reset, en,
              input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q);

             always_ff@(posedge clk, posedge reset)
                    if (reset) q<=0;
                    else if (en) q<=d;

endmodule

// parametrized DFF
module flopr #(parameter WIDTH = 10)
             (input logic clk, reset,
              input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q);

             always_ff@(posedge clk, posedge reset)
                    if (reset) q<=0;
                    else q <= d;

endmodule

// parametrized counter
module counter #(parameter WIDTH = 6)
              (input logic clk,
               input logic reset,
               output logic [WIDTH-1:0] q);

             always_ff@(posedge clk, posedge reset)
                    if(reset) q<=0;
                    else q <= q + 1;

endmodule
```

Appendix B: heartRateFinder State Transition Diagram



outside = 30 < interHR < 250

close = interHR is within ±25% of average of first 3 heart rates

interHR = 2,400,000,000/pulseCount

Appendix C: C Code

```c
// E155 Final Project: Heart Rate-Controlled Music
// Senghor Joseph and Christine Goins
// sjoseph@hmc.edu    cgoins@hmc.edu
// ampisw.c reads a target heart rate from pins connected to switches
// and requests and reads the calculated heart rate from the FPGA
// and calculates a speed factor from the difference to change the speed of the song
// Fur Elise using square waves of varying frequencies and durations
#include "start.h"

// Pitch in Hz, duration in ms
const int notes[][2];

void playMusic(int pitch, int duration);
float speedFactor(int target, int avg_hr);

// playMusic generates squares waves with the given frequencies and durations to play notes
void playMusic(int pitch, int duration){
  // plays a note given pitch and duration

  unsigned int wavemicros; // 1/2 the period of the note

  if(pitch!=0) {
    wavemicros = 1000000/pitch/2; // converts to us and 1/2 period
  }

  unsigned int durationmicros = duration*1000; // convert from ms to us

  // while loop for length of note (duration) (timer 1)
  sys_timer[4]=sys_timer[1] + durationmicros; // C1 = CL0 + durationmicros
  sys_timer[0] &= 0b0010; // M1=0
  while(!(sys_timer[0] &= 0b0010)) { // while flag M1 is low
    // making wave with desired frequency (timer 2)
    if(pitch == 0) { // pitch=0 indicates a rest
      digitalWrite(23,0);
    } else {
      delayMicroseconds(wavemicros);
      digitalWrite(23,1);
      delayMicroseconds(wavemicros);
      digitalWrite(23,0);
    }
  }
}

// speedFactor uses the difference between the target and average heart rates to calculate
// a speedFactor sf used to change the speed of the music
float speedFactor(int target, int avg_hr) {

  float sf;
  if((avg_hr >= target-10) & (avg_hr < target+10)) { // within +/-10 range
    sf = 1;
  } else if((avg_hr >= target-20) & (avg_hr < target-10)) { //10-20 below
    sf = 1.25;
  } else if((avg_hr >= target-30) & (avg_hr < target-20)) { //20-30 below
    sf = 1.5;
  } else if((avg_hr >= target-40) & (avg_hr < target-30)) { //30-40 below
    sf = 1.75;
  } else if((avg_hr < target+20) & (avg_hr >= target+10)) { //10-20 above
    sf = 0.8;
  } else if((avg_hr < target+30) & (avg_hr >= target+20)) { //20-30 above
    sf = 0.67;
  } else if((avg_hr < target+40) & (avg_hr >= target+30)) { //30-40 above
    sf = 0.57;
  } else if(avg_hr < target-40) { // more than 40 below
    sf = 2;
  } else if(avg_hr >= target+40) {
    sf = 0.5;
  } else {
    sf = 0.42;
  }

  return sf;
}



void main(void){

  // HTML header
  printf("%s%c%c\n",
              "Content-Type:text/html;charset=iso-8859-1",13,10);

  int fclk;
  float sf = 1;
  int heartrate;
  int count = 0;
  int avg_hr;
  int add_hr = 0;
  int i = 0;
  int target;
  int bit7, bit6, bit5, bit4, bit3, bit2, bit1, bit0;

  pioInit();
  pinMode(23, OUTPUT);
  pinMode(21, INPUT);
  pinMode(18, INPUT);
  pinMode(17, INPUT);
  pinMode(16, INPUT);
  pinMode(12, INPUT);
  pinMode(6, INPUT);
  pinMode(5, INPUT);
  pinMode(4, INPUT);

  // set up SPI
  fclk = 122000; // set pi_clk for FPGA
  spiInit(fclk, 00); // set phase and pol to 0

  // reads measured and target heart rates and uses speedFactor to calculate the speed by
  // which to change the music and playMusic to play each note of the song
  while(notes[i][1]!=0){ // duration of 0 indicates end of song

    if(count==9){ // recalculates speed factor every 10 notes
      heartrate = (int) SPIsendReceive(0b00000000);
      add_hr += heartrate;
      avg_hr = add_hr/10; // calculate average heart rate of last 10
      sf = speedFactor(target, avg_hr);
      playMusic(notes[i][0], notes[i][1]/sf);
      count = 0;
      add_hr = 0;
      // read target heart rate
      bit7 = digitalRead(21);
      bit6 = digitalRead(18);
      bit5 = digitalRead(17);
      bit4 = digitalRead(16);
      bit3 = digitalRead(12);
      bit2 = digitalRead(6);
      bit1 = digitalRead(5);
      bit0 = digitalRead(4);
      target = (int) (bit7<<7) | (bit6<<6) | (bit5<<5) | (bit4<<4) | (bit3<<3) | (bit2<<2) |
(bit1<<1) | bit0;
      i++;
       printf("<p>Target Heart Rate: %d\n\n\nCurrent Heart Rate: %d</p>",target, heartrate);

    } else {
      heartrate = (int) SPIsendReceive(0b00000000);
      playMusic(notesi[i][0], notes[i][1]/sf);
      add_hr += heartrate;
      count ++;
      i++;
    }
  }
}
```

Appendix D: HTML Code

```
<html>
	<head></head>
	<body>
		<img
src="http://www.clipartkid.com/images/208/music-notes-heart-beat-clipart-panda-free-clipart-im
ages-erFXrz-clipart.jpg">
		<header>
			<h1>Heart Rate & Music</h1>
			<h2>Christine Goins and Senghor Joseph</h2><br>
			<h3>Enter your desired [Target Heart Rate] on the DIP switches </h3>
			<h3>View your current heart rate on the LED array below</h3>

			<iframe src="cgi-bin/ampisw" width="300px" height="200px"></iframe>

		</header>
	</body>
</html>
```