Whistle Pongbat

Peter Capraro
Michael Hankin
Anand Rajeswaran

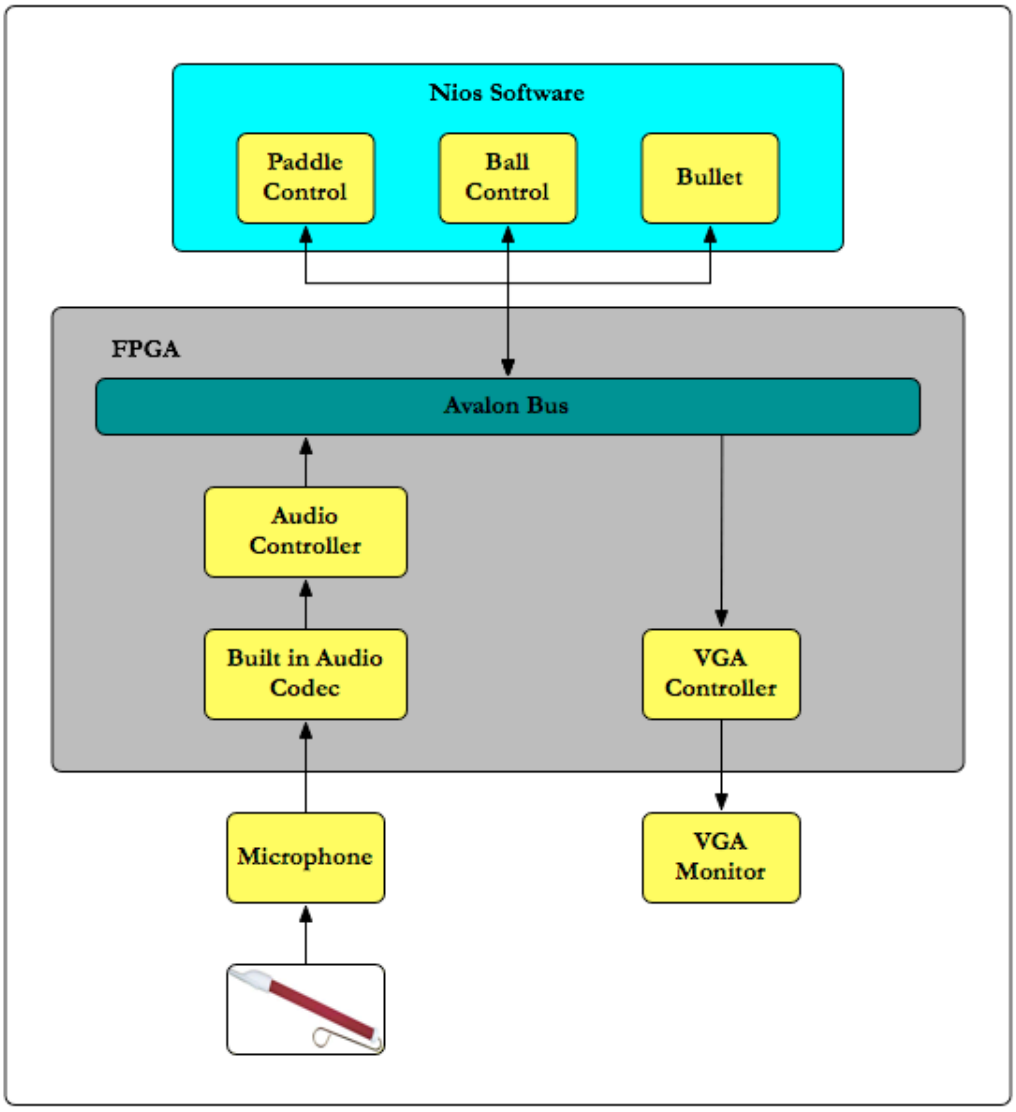
1. Introduction

Whistle Pongbat is the reincarnation of a classic video game, with a new twist. We implemented a pong game using the VGA monitor as our output, but rather than the traditional buttons or joysticks to control the paddle, we decided to foray into the world of audio and use a microphone as the input to drive the control of our paddle. To control the paddle, we wanted to use the frequency of the audio input. A high frequency of input would drive the paddle upwards whereas a low frequency would push it downwards. However, this design was dependent on being able to calculate a consistent frequency from the input. To avert this potential obstacle, we decided to use a slide-whistle as our audio input. In addition to providing a consistent pitch, this would also provide us with a near perfect sine wave to simplify the frequency calculation and also to provide a constant, bounded range of frequencies. On the way to completing a frequency controlled paddle, our first aim was to implement a noise controlled paddle. In the presence of no audio input, the paddle constantly moves upwards, and while sound is

detected, it moves downwards. Upon reaching this milestone, and extensively testing it (it was fun to play), we decided to include the noise-controlled paddle as an option, and create a simple mechanism for the user to choose a paddle control option. When the game starts, the paddle is in the middle of the screen. By whistling at a high frequency so that the paddle reaches the top of the screen, the noise controlled paddle option is chosen. Whistling at a low frequency to send the paddle to the bottom of the screen invokes the frequency controlled paddle option.

Given that there is only one audio line in on the FPGA, making a two-player game was not feasible. Instead, we adapted the game to single player game, with a set of stagnant blocks in a vertical row along the right edge acting as a "computer". By hitting the ball off the user paddle a set amount of times, they build up and fire a bullet that can destroy one of the blocks on the other end. After destroying a block (or multiple blocks), the user is able to win a point if they can hit the ball through the vacated hole. The "computer" can win a point if the ball gets to the left side of the screen without making contact with the paddle. The first to reach five points wins the game, at which point it returns to the start screen where the paddle control option for the next game can be chosen.

2. Design

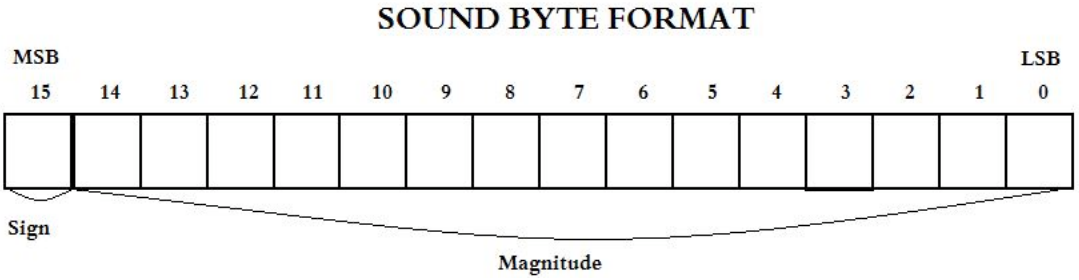


2.1 Architecture

The overall design architecture is shown above. On the hardware side of the block diagram, there are two important pieces: the audio controller and the VGA controller. The audio controller connects the whistle and microphone analog input to the audio CODEC from the chip and uses the audio to digital converter to write digital data to the Avalon Bus. Meanwhile, the VGA controller reads data about paddle, ball, bullet, computer blocks, and score from the Avalon Bus and uses this to paint the VGA monitor. Therefore, to connect the two of these hardware segments, the software's job is to insert the digital sound byte from the audio controller into our frequency algorithm to control the paddle position. In addition, it is responsible for controlling the movement of the ball and the bullet, as well as maintaining the score. Each of these hardware and software components will be described in detail in the following sections.

2.2 Hardware - Audio Controller

The audio controller uses the Wolfson WM8731 audio CODEC built into the board to translate the analog input from the microphone into a digital signal. From this top-level signal from the board, we used the de2_wm8731_audio_in module from the MindTunes group of 2008. The purpose of this module is to convert the raw digital data into a 16 bit sound byte adhering to the .WAV format (explained below in the diagram). It does this by splitting the audio clock and storing every 16 bits received from the audio to digital converter in an array. On each rising edge of the split clock, this array has been filled, so it is output to the Avalon Bus. Finally, to relieve the software from having to cycle through 16 memory slots to read each sound byte, the sound is placed bit by bit into one address of the register. The result is that it outputs to the software an integer representation of the 16 bits, ranging from 0 to 65,535.



The most significant bit of the sound byte is the sign bit. If this bit is 0 (positive number), then the resulting integer sent to the software ranges from 0 to 32,767 in order of increasing magnitude. If it is 1 (negative), then the integer ranges from 32,768 to 65,535 in decreasing magnitude, as a two's complement system is used. This relation of the physical bits in the sound byte to the integer passed to the software will be important for the frequency algorithm implemented in the software.

2.3 Hardware - VGA Controller

The VGA controller receives information about the game from the software to paint the screen appropriately. The lab 3 VGA raster for displaying a bouncing ball was our basis for this component. We added registers to track the other game components necessary to correctly draw the VGA monitor and the necessary logic to determine which pixels should be drawn. A full list of registers is below.

Address	Data
0x00	Paddle Height
0x01	Ball X Coordinate
0x02	Ball Y Coordinate
0x03	User Score
0x04	Computer Score
0x05	Block 1 (Top Block)
0x06	Block 2
0x07	Block 3
0x08	Block 4
0x09	Block 5
0x0A	Block 6 (Bottom Block)
0x0B	Bullet Indicator
0x0C	Bullet Charging Counter
0x0D	Bullet X coordinate
0x0E	Bullet Y coordinate

2.4 Software - Paddle Controller

As described in the introduction, the paddle controller implements either the noise-controlled paddle or the frequency-controlled paddle. In the case of the noise-controlled paddle, the operation is simple. The paddle automatically drifts upwards at a constant speed if silence is detected. Silence, for our purposes, is defined as 50 consecutive sound bytes of a negligible magnitude. The magnitude of a sound byte is determined based on its sign (the integer representation of the sound byte is more completely described in section 2.2). If it is positive, then the magnitude is simply the value of the integer passed in from the audio controller. If it is negative, then the magnitude is 65,535 minus the integer value of the sound byte. This is not intuitive because a negative number uses the first bit as its sign bit and a two's complement representation in the .WAV format. Instead of having to deal with this arithmetic in hardware, we simply convert the bits to an integer as though they were a binary string. The resulting integer is packed into a register for use in the software. The exact magnitude at which we determined sound was negligible was determined by trial and error.

In the case of the frequency controlled paddle, each incoming sound byte is sent into a frequency function. The purpose of this function is not to provide an absolute frequency, as only a relative frequency metric is enough for our purposes. We use a simple method of counting the number of samples between sign changes as an estimate of its period. This method, derived from http://lukeallen.org/whistleswitch.html, counts the number of consecutive negative sound bytes, then counts the number of positive sound bytes, storing each of these as it goes. It keeps track of the previous 22 counts in an array and uses the average of these 22 counts as an estimate of the period. As the period is the inverse of the frequency, it is enough for our context to treat a high period as a low frequency and vice versa, without actually calculating the frequency.

2.5 Software – Ball Controller

The ball movement is controlled by two variables: the speed, in pixels/iteration and the direction, in degrees. From the direction, the sine and cosine functions can determine the ratio of movement in the vertical and horizontal directions respectively. Multiplying the speed by these ratios determines the magnitude of movement in either direction. A key to this process is that the position of the ball is kept as a double in the software and always cast to an integer to be written to the bus. Each iteration, the ball only moves a few pixels at a time (or less). If the value were kept as an integer, the movement would be rounded each time and would have a noticeable impact on its direction and magnitude.

The other aspect of ball movement is the handling of its collisions. If the ball reaches either the top or bottom of the screen, redirection is straightforward by simply changing the direction attribute to signify the rebound. If the ball collides with the blocks on the right side of the screen, a similar redirection is performed. However, if a ball reaches the right side of the screen and a bullet has already taken out the block that would have reflected the ball, the user wins the point. The score is updated and a new point begins. On the left side of the screen, the only way a ball can collide is if it makes contact with the paddle. If there is no contact, the "computer" wins the point. Alternatively, if the paddle is in position to make contact, redirection is based on the position of contact on the paddle. The paddle surface uses a gradient so that a ball which makes contact towards the top of the paddle rebounds in a more upward direction and a ball which makes contact at the bottom of the paddle rebounds in a more downward pattern.

2.6 Software – Bullet Controller

The bullet system is controlled automatically whenever the ball makes contact with the paddle. Every time this happens, the bullet is partially charged. This charging is visible on the screen as the bullet forms in the middle of the paddle. After three times, the bullet is fully charged, but still attached to the paddle. Upon the fourth hit, the bullet is automatically fired straight across the screen at a constant speed. When the bullet finally reaches the right side of the screen, it destroys the block it makes contact with, if the block has not already been destroyed. If it hits the boundary of two blocks, it destroys both. As soon as the bullet is fired from the paddle, a new bullet begins charging.

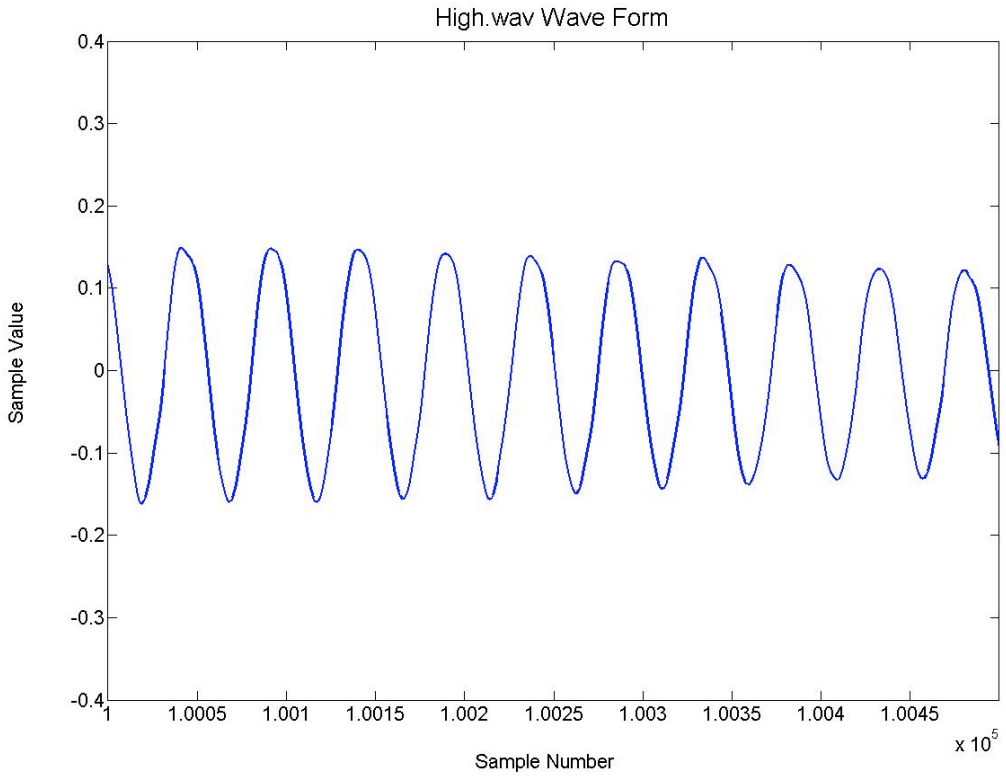
2.7 Physical Peripheral – Slide Whistle Controller

The slide whistle was combined with the microphone to create a controller for this game. This was fabricated from the Mechanical Engineering rapid prototype machine, and served the purpose of keeping the whistle a constant distance from the microphone. This also alleviated the burden of the user having to hold the microphone and the whistle at the same time, while operating the slide to alter the frequency of the whistle.

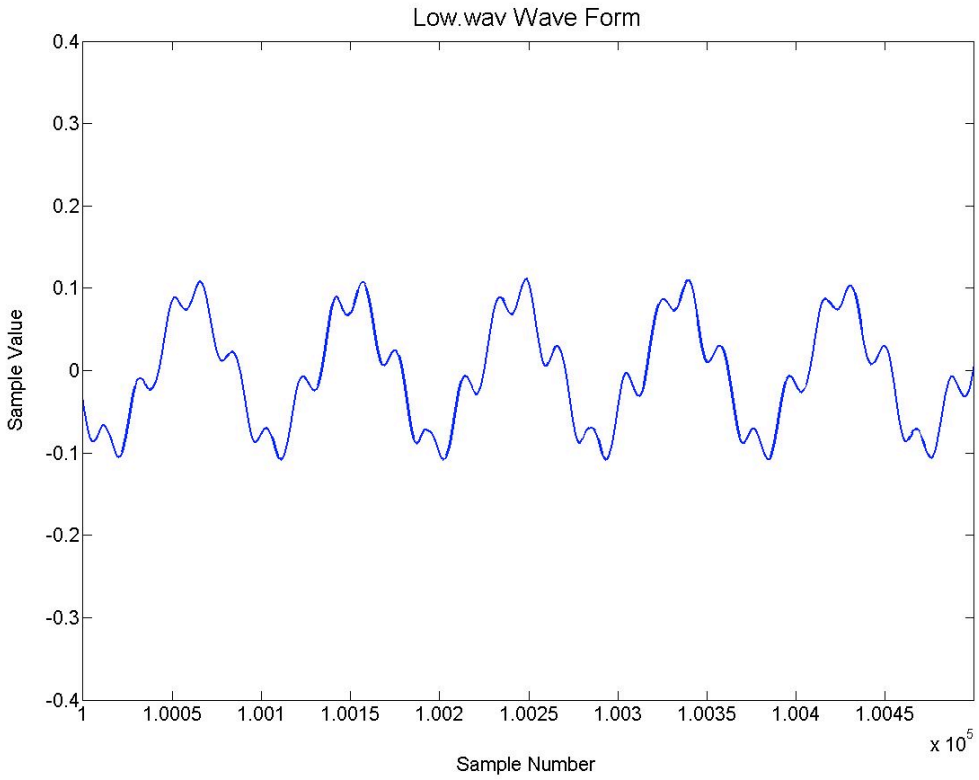
3. Design Issues

3.1 Frequency Algorithm

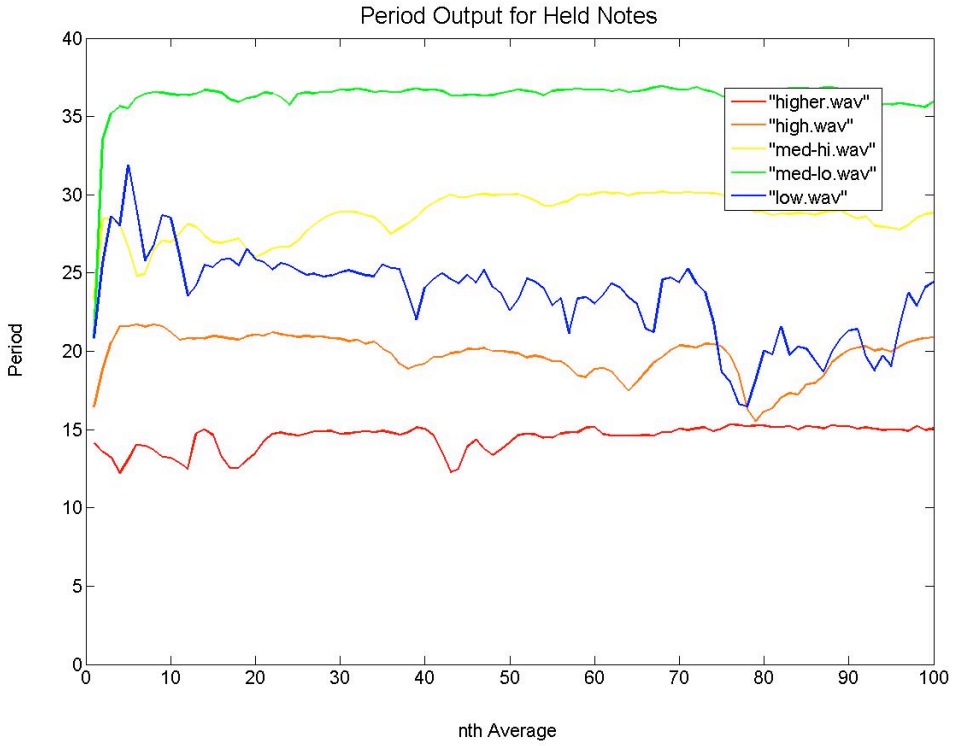
One of our initial goals and challenges was to come up with an algorithm that could take a constant stream of data in and regularly output some measure of frequency. To accomplish this, we first began working in MATLAB due to its convenient built in features for analyzing and viewing large amounts of data. To simulate the data input, we recorded WAV files of various slide whistle tones. We first looked at the raw data to determine the range of amplitudes and frequencies. Much to our delight, for the majority of the pitch range, the data took on a fairly decent looking sin wave, shown below.



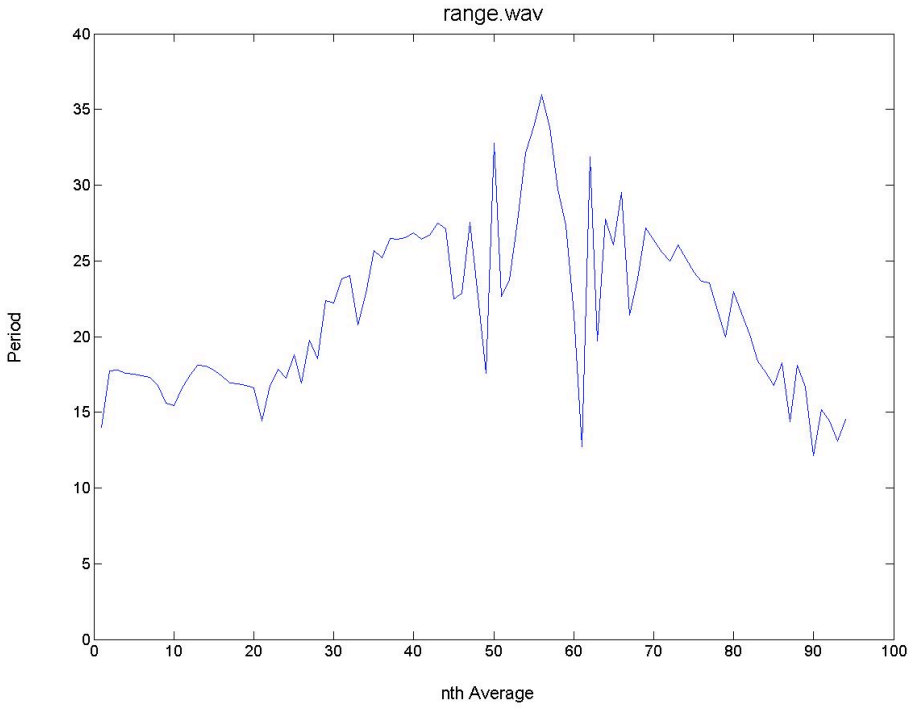
However, we were all saddened to find at the low range of the whistle, these uniform waves broke down, showing multiple frequencies.



We decided that despite the irregular signal, we could probably find an accurate measurement of the period by counting the number of samples between zero crossings. The algorithm works by reducing the signal to a square wave between 0 and 1. It iterates through the WAV files, considering one sample at a time. If the sample is positive, it assumes an "on" state and begins (or continues) counting the number of consecutive hi samples. If a sample coming in is negative, it assumes an "off" state and begins (or continues) counting the number of consecutive lo samples. In the case where a sample is hi and the state is off, or a sample is lo and the state is on, it toggles the state. We also experimented with various tolerances for "noise" in the signal (meaning if there is a state change for only a single or small number of samples). Finally, we accounted for a lack of sound coming in. Since we couldn't just ignore values below a certain amplitude (since a sin wave also includes these values) we did this by counting samples that were lower than a certain value, and considering it silence after the count reached a certain number (just larger than the period of the lowest frequency samples). The results as shown below, were fairly consistent for the majority of the range.



As expected, the algorithm broke down to some degree causing the blue curve to fall out of place on the frequency rainbow shown above. Furthermore, upon rapidly changing pitches, rather than holding, the graph would produce spikes in certain areas. Below shows the period output for a rapid slide from high to low to high pitch.



Despite these flaws in the algorithm, we decided it would be adequate for our intentions. We considered moving ahead with two options. In the first, the frequency would correspond to a desired paddle destiny rather than the current location. Then the paddle would slowly travel towards that location. This would require the player to hold notes out for longer, and also average out small discrepancies in frequency readings. The second would be modeled after traditional Pong control, which is simple up/down control. In this case, a frequency above a certain value would cause the paddle to go up, whereas a frequency below that value would cause it to go down.

3.2 Audio Connection

The most daunting obstacle of our project to overcome was simply to configure the audio connection to the FPGA. Using a combination of the audio portion of lab 3 (provided by Professor Edwards) and the MindTunes project (4840 project from 2008), we were finally able to configure a VHDL file to convert the output of the audio CODEC into a sound byte of the WAV format. Also, we were able to create an audio controller, in order to pass this WAV sound byte to the audio bus for manipulation by the software. However, at this point, we were still unable to locate any noise with a microphone plugged into the MIC port of the FPGA. After many different configurations of our audio controller were unable to fix this and calling the FPGA board many inappropriate names was also unable to fix this, we decided to test the microphone in the LINE IN port. As this went completely against our intuition, and made no common sense, it happened to work perfectly. Although to this day we remain unsure as to why this occurred, we believe it is because we used a microphone normally used for electrical engineering purposes with a converter attached. It is likely that this setup, with the converter, supplies us with a different signal than a normal microphone created for the MIC line.

3.3 Oscillating Frequency

The original idea of our project was to have the paddle be sensitive to the pitch of the user's voice, forcing users of our game to make as many embarrassing noises as possible. However, we decided to limit the audio input to that of a slide whistle, due to concerns as to whether we could devise an algorithm to detect a consistent pitch from a human voice. Therefore, upon affirming that our frequency algorithm was operating correctly, it took us by surprise that even a slide whistle was not producing a consistent frequency.

Through extensive analysis of results of our algorithm, we determined that in general our output was changing correctly as the frequency changed. High frequency input produced, on average, lower results when calculating the period and low frequency input, on average, produced higher results. However, even with a constant pitch from the slide whistle, the result of our algorithm did not stay closely centered around one value. The frequent and wide oscillation made the paddle movement unpredictable, so we had to change our paddle movement mechanism. If the period reaches either a high or low boundary, (far enough apart that a low period note would not usually ever generate a period value above the high boundary and vice versa) the direction of paddle motion is set in the respective direction. This direction remains unchanged unless the other boundary is reached or the algorithm described for the noise-controlled paddle detects silence. Even though the frequency was not consistently above or below the required boundary, we assumed that the reaching of the boundary was always indicative of the user's intended frequency. When the other extreme boundary was reached, even just once, we could safely assume the user was attempting to change the direction of the paddle, because of the large disparity between the boundaries. By this method, we were able to translate substantial oscillation of the frequency into discrete paddle movements.

4. Summary and Lessons Learned

At the end of the semester, we were able to accomplish the goals we had set for ourselves, even though we began without confidence in our ability to do this. Our group was very inexperienced in terms of understanding the FPGA board and hardware to software communication, given our diverse backgrounds as a mechanical engineer, an applied mathematician, and sadly, even a computer

engineer. Therefore, our greatest obstacle was configuring the board correctly and setting up the communication between the hardware and software. Given this, we tended to overestimate ourselves with the milestones we attempted to reach, and did not plan adequately to test every small step of progress. As should have been expected, when all the components had been created and the program was still not working, this made it difficult to pinpoint our errors.

Also, given that we had never done anything involving the audio ports of the FPGA before, we learned the importance of looking elsewhere for examples and guidance. With this, we had to learn the importance of reading past the comments of others' code. We were held up with a certain error for a long time before we realized that one signal from code we re-used was not acting as they had described it would.

Overall we were very pleased with the final result we were able to attain, and most pleased knowing we had learned most every skill that went into the project throughout the course of the...course. Most of all, we look forward to honing our slide whistle skills as we seek to master (finally beat just one time) Whistle Pongbat.

5. Source Code

5.1 lab3_vga.vhd

Top-Level VHDL File – Adapted from Lab 3

```
 --
-- DE2 top-level module that includes the simple VGA raster generator
--
-- Stephen A. Edwards, Columbia University, sedwards@cs.columbia.edu
--
-- From an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lab3_vga is

 port (



-- Clocks

   CLOCK_27,                        -- 27 MHz
   CLOCK_50,                        -- 50 MHz
   EXT_CLOCK : in std_logic;               -- External Clock

   -- Buttons and switches

   KEY : in std_logic_vector(3 downto 0);       -- Push buttons
   SW : in std_logic_vector(17 downto 0);       -- DPDT switches

   -- LED displays
```

```
    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 -- 7-segment displays
      : out std_logic_vector(6 downto 0);
    LEDG : out std_logic_vector(8 downto 0);       -- Green LEDs
    LEDR : out std_logic_vector(17 downto 0);      -- Red LEDs

    -- RS-232 interface

    UART_TXD : out std_logic;                  -- UART transmitter
    UART_RXD : in std_logic;                   -- UART receiver

    -- IRDA interface

--    IRDA_TXD : out std_logic;                  -- IRDA Transmitter
    IRDA_RXD : in std_logic;                   -- IRDA Receiver

    -- SDRAM

    DRAM_DQ : inout std_logic_vector(15 downto 0); -- Data Bus
    DRAM_ADDR : out std_logic_vector(11 downto 0); -- Address Bus
    DRAM_LDQM,                                 -- Low-byte Data Mask
    DRAM_UDQM,                                 -- High-byte Data Mask
    DRAM_WE_N,                                 -- Write Enable
    DRAM_CAS_N,                                -- Column Address Strobe
    DRAM_RAS_N,                                -- Row Address Strobe
    DRAM_CS_N,                                 -- Chip Select
    DRAM_BA_0,                                 -- Bank Address 0
    DRAM_BA_1,                                 -- Bank Address 0
    DRAM_CLK,                                  -- Clock
    DRAM_CKE : out std_logic;                  -- Clock Enable

    -- FLASH

    FL_DQ : inout std_logic_vector(7 downto 0);      -- Data bus
    FL_ADDR : out std_logic_vector(21 downto 0);  -- Address bus
    FL_WE_N,                                   -- Write Enable
    FL_RST_N,                                  -- Reset
    FL_OE_N,                                   -- Output Enable
    FL_CE_N : out std_logic;                   -- Chip Enable

    -- SRAM

    SRAM_DQ : inout std_logic_vector(15 downto 0); -- Data bus 16 Bits
    SRAM_ADDR : out std_logic_vector(17 downto 0); -- Address bus 18 Bits
    SRAM_UB_N,                                 -- High-byte Data Mask
    SRAM_LB_N,                                 -- Low-byte Data Mask
    SRAM_WE_N,                                 -- Write Enable
    SRAM_CE_N,                                 -- Chip Enable
    SRAM_OE_N : out std_logic;                 -- Output Enable

    -- USB controller

    OTG_DATA : inout std_logic_vector(15 downto 0); -- Data bus
    OTG_ADDR : out std_logic_vector(1 downto 0);    -- Address
    OTG_CS_N,                                  -- Chip Select
    OTG_RD_N,                                  -- Write
    OTG_WR_N,                                  -- Read
```

```
OTG_RST_N,                          -- Reset
OTG_FSPEED,                 -- USB Full Speed, 0 = Enable, Z = Disable
OTG_LSPEED : out std_logic;    -- USB Low Speed, 0 = Enable, Z = Disable
OTG_INT0,                           -- Interrupt 0
OTG_INT1,                           -- Interrupt 1
OTG_DREQ0,                           -- DMA Request 0
OTG_DREQ1 : in std_logic;                 -- DMA Request 1
OTG_DACK0_N,                         -- DMA Acknowledge 0
OTG_DACK1_N : out std_logic;               -- DMA Acknowledge 1

-- 16 X 2 LCD Module

LCD_ON,                -- Power ON/OFF
LCD_BLON,                -- Back Light ON/OFF
LCD_RW,                -- Read/Write Select, 0 = Write, 1 = Read
LCD_EN,                -- Enable
LCD_RS : out std_logic;    -- Command/Data Select, 0 = Command, 1 = Data
LCD_DATA : inout std_logic_vector(7 downto 0); -- Data bus 8 bits

-- SD card interface

SD_DAT,                -- SD Card Data
SD_DAT3,                -- SD Card Data 3
SD_CMD : inout std_logic;   -- SD Card Command Signal
SD_CLK : out std_logic;    -- SD Card Clock

-- USB JTAG link

TDI,                   -- CPLD -> FPGA (data in)
TCK,                   -- CPLD -> FPGA (clk)
TCS : in std_logic;        -- CPLD -> FPGA (CS)
TDO : out std_logic;       -- FPGA -> CPLD (data out)

-- I2C bus

I2C_SDAT : inout std_logic; -- I2C Data
I2C_SCLK : out std_logic;   -- I2C Clock

-- PS/2 port

PS2_DAT,                -- Data
PS2_CLK : in std_logic;    -- Clock

-- VGA output

VGA_CLK,                          -- Clock
VGA_HS,                          -- H_SYNC
VGA_VS,                          -- V_SYNC
VGA_BLANK,                          -- BLANK
VGA_SYNC : out std_logic;                -- SYNC
VGA_R,                          -- Red[9:0]
VGA_G,                          -- Green[9:0]
VGA_B : out unsigned(9 downto 0);            -- Blue[9:0]

--  Ethernet Interface
```

```
    ENET_DATA : inout std_logic_vector(15 downto 0);    -- DATA bus 16Bits
    ENET_CMD,          -- Command/Data Select, 0 = Command, 1 = Data
    ENET_CS_N,                                  -- Chip Select
    ENET_WR_N,                                   -- Write
    ENET_RD_N,                                  -- Read
    ENET_RST_N,                                  -- Reset
    ENET_CLK : out std_logic;                        -- Clock 25 MHz
    ENET_INT : in std_logic;                        -- Interrupt

    -- Audio CODEC

    AUD_ADCLRCK : inout std_logic;                    -- ADC LR Clock
    AUD_ADCDAT : in std_logic;                      -- ADC Data
    AUD_DACLRCK : inout std_logic;                    -- DAC LR Clock
    AUD_DACDAT : out std_logic;                     -- DAC Data
    AUD_BCLK : inout std_logic;                    -- Bit-Stream Clock
    AUD_XCK : out std_logic;                      -- Chip Clock

    -- Video Decoder

    TD_DATA : in std_logic_vector(7 downto 0);  -- Data bus 8 bits
    TD_HS,                                  -- H_SYNC
    TD_VS : in std_logic;                       -- V_SYNC
    TD_RESET : out std_logic;                    -- Reset

    -- General-purpose I/O

    GPIO_0,                                  -- GPIO Connection 0
    GPIO_1 : inout std_logic_vector(35 downto 0) -- GPIO Connection 1
    );

end lab3_vga;

architecture datapath of lab3_vga is


   component de2_wm8731_audio_in is
    port (
       clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
       reset_n : in std_logic;
       data_out : out std_logic_vector(15 downto 0);
       audio_req : out std_logic;

       -- Audio interface signals
       AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
       AUD_ADCDAT   : in  std_logic;   --    Audio CODEC ADC Data
       AUD_BCLK     : inout std_logic  --    Audio CODEC Bit-Stream Clock
    );
   end component;

signal audio_clock : unsigned(1 downto 0) := "00";
signal audio_request : std_logic;
signal clk25 : std_logic := '0';
 signal counter : unsigned(15 downto 0);
signal reset_n : std_logic;
signal audio_data_in : std_logic_vector (15 downto 0);
```

```
 component de2_i2c_av_config is
 port (
   iCLK : in std_logic;
   iRST_N : in std_logic;
   I2C_SCLK : out std_logic;
   I2C_SDAT : inout std_logic
 );
 end component;



begin
LEDR(17) <= '1';
 LEDR(16) <= '1';

 i2c : de2_i2c_av_config port map (
   iCLK     => CLOCK_50,
   iRST_n   => '1',
   I2C_SCLK => I2C_SCLK,
   I2C_SDAT => I2C_SDAT
 );


 process (CLOCK_50)
 begin
   if rising_edge(CLOCK_50) then
     clk25 <= not clk25;
--      audio_clock <= audio_clock + "1";
   end if;
 end process;
--  AUD_XCK <= audio_clock(1);
--  V1: entity work.de2_vga_raster port map (
 --   reset => '0',
--    clk => clk25,
--    VGA_CLK => VGA_CLK,
--    VGA_HS => VGA_HS,
--    VGA_VS => VGA_VS,
--   VGA_BLANK => VGA_BLANK,
--    VGA_SYNC => VGA_SYNC,
--    VGA_R => VGA_R,
--    VGA_G => VGA_G,
--    VGA_B => VGA_B,
--  read => '0',
--write => '0' ,
--chipselect => '0',
--address => "00000", --: in unsigned(4 downto 0);
-- readdata => "0000000000000000", --: out unsigned(15 downto 0);
--writedata => "0000000000000000" -- : in unsigned(15 downto 0);



--);
```

```
--------

 process (CLOCK_50)
 begin
  if rising_edge(CLOCK_50) then
    if counter = x"ffff" then
      reset_n <= '1';
    else
      reset_n <= '0';
      counter <= counter + 1;
    end if;
  end if;
 end process;

 process (CLOCK_50)
 begin
  if rising_edge(CLOCK_50) then
    audio_clock <= audio_clock + "1";
  end if;
 end process;

 AUD_XCK <= audio_clock(1);



  ADC : de2_wm8731_audio_in
  port map (
     clk => audio_clock(1),
     reset_n => reset_n,
     data_out => audio_data_in,
     audio_req => audio_request,
     AUD_ADCLRCK => AUD_ADCLRCK,
     AUD_ADCDAT => AUD_ADCDAT,
     AUD_BCLK => AUD_BCLK
  );

 nios : entity work.nios_system port map (
  clk                       => CLOCK_50,
    clk25                                                          => clk25,
  reset_n                    => reset_n,

  VGA_CLK_from_the_raster => VGA_CLK,
  VGA_HS_from_the_raster => VGA_HS,
  VGA_VS_from_the_raster => VGA_VS,
  VGA_BLANK_from_the_raster => VGA_BLANK,
  VGA_SYNC_from_the_raster => VGA_SYNC,
  std_logic_vector(VGA_R_from_the_raster) => VGA_R(9 downto 0),
  std_logic_vector(VGA_G_from_the_raster) => VGA_G(9 downto 0),
  std_logic_vector(VGA_B_from_the_raster) => VGA_B(9 downto 0),

  SRAM_ADDR_from_the_sram      => SRAM_ADDR,
  SRAM_CE_N_from_the_sram      => SRAM_CE_N,
  SRAM_DQ_to_and_from_the_sram => SRAM_DQ,
```

```
    SRAM_LB_N_from_the_sram      => SRAM_LB_N,
    SRAM_OE_N_from_the_sram      => SRAM_OE_N,
    SRAM_UB_N_from_the_sram      => SRAM_UB_N,
    SRAM_WE_N_from_the_sram      => SRAM_WE_N,

      audio_data_in_to_the_audioslave => audio_data_in,
      audio_request_to_the_audioslave => audio_request
--    AUD_ADCLRCK_to_and_from_the_audioslave => AUD_ADCLRCK,
--    AUD_ADCDAT_to_the_audioslave => AUD_ADCDAT,
--    AUD_DACLRCK_to_and_from_the_audioslave => AUD_DACLRCK,
--    AUD_DACDAT_from_the_audioslave => AUD_DACDAT,
--    AUD_BCLK_to_and_from_the_audioslave => AUD_BCLK,
--    AUD_XCK_from_the_audioslave => AUD_XCK
--    I2C_SDAT_to_and_from_the_audioslave => I2C_SDAT,
--    I2C_SCLK_from_the_audioslave => I2C_SCLK


    );



------

  HEX7     <= "0001100"; -- Leftmost
  HEX6     <= "0100011";
  HEX5     <= "0101011";
  HEX4     <= "0010000";
  HEX3     <= "0000011";
  HEX2     <= "0001000";
  HEX1     <= "0111001";
  HEX0     <= (others => '1');          -- Rightmost
  LEDG     <= (others => '1');
--  LEDR     <= (others => '1');
  LCD_ON   <= '1';
  LCD_BLON <= '1';
  LCD_RW <= '1';
  LCD_EN <= '0';
  LCD_RS <= '0';

  SD_DAT3 <= '1';
  SD_CMD <= '1';
  SD_CLK <= '1';

--  SRAM_DQ <= (others => 'Z');
--SRAM_ADDR <= (others => '0');
-- SRAM_UB_N <= '1';
-- SRAM_LB_N <= '1';
-- SRAM_CE_N <= '1';
--SRAM_WE_N <= '1';
-- SRAM_OE_N <= '1';
```

```
UART_TXD <= '0';
DRAM_ADDR <= (others => '0');
DRAM_LDQM <= '0';
DRAM_UDQM <= '0';
DRAM_WE_N <= '1';
DRAM_CAS_N <= '1';
DRAM_RAS_N <= '1';
DRAM_CS_N <= '1';
DRAM_BA_0 <= '0';
DRAM_BA_1 <= '0';
DRAM_CLK <= '0';
DRAM_CKE <= '0';
FL_ADDR <= (others => '0');
FL_WE_N <= '1';
FL_RST_N <= '0';
FL_OE_N <= '1';
FL_CE_N <= '1';
OTG_ADDR <= (others => '0');
OTG_CS_N <= '1';
OTG_RD_N <= '1';
OTG_RD_N <= '1';
OTG_WR_N <= '1';
OTG_RST_N <= '1';
OTG_FSPEED <= '1';
OTG_LSPEED <= '1';
OTG_DACK0_N <= '1';
OTG_DACK1_N <= '1';

TDO <= '0';

ENET_CMD <= '0';
ENET_CS_N <= '1';
ENET_WR_N <= '1';
ENET_RD_N <= '1';
ENET_RST_N <= '1';
ENET_CLK <= '0';

TD_RESET <= '0';

--I2C_SCLK <= '1';

-- Set all bidirectional ports to tri-state
DRAM_DQ     <= (others => 'Z');
FL_DQ       <= (others => 'Z');
SRAM_DQ     <= (others => 'Z');
OTG_DATA    <= (others => 'Z');
LCD_DATA    <= (others => 'Z');
SD_DAT      <= 'Z';
--I2C_SDAT    <= 'Z';
ENET_DATA   <= (others => 'Z');
--AUD_ADCLRCK <= 'Z';
--AUD_DACLRCK <= 'Z';
--AUD_BCLK    <= 'Z';
GPIO_0      <= (others => 'Z');
```

```
        GPIO_1      <= (others => 'Z');

      end datapath;
```

5.2 de2_wm8731_audio_in.vhd

Module for converting digital audio (from chip) to WAV format – Adapted from MindTunes group (2008)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- de2_wm8731_audio_in : generate clock and get the samples from device

entity de2_wm8731_audio_in is
port (
   clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
   reset_n : in std_logic;
   data_out : out std_logic_vector(15 downto 0);
   audio_req : out std_logic;

   -- Audio interface signals
         AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
   AUD_ADCDAT   : in  std_logic;   --    Audio CODEC ADC Data
   AUD_BCLK     : inout std_logic  --    Audio CODEC Bit-Stream Clock
  );
end  de2_wm8731_audio_in;


architecture Behavioral of  de2_wm8731_audio_in is

   signal lrck : std_logic;
   signal bclk : std_logic;
   signal xck : std_logic;

   signal lrck_divider : std_logic_vector (7 downto 0);
   signal bclk_divider : std_logic_vector (3 downto 0);

   signal set_bclk : std_logic;
   signal set_lrck : std_logic;
   signal lrck_lat : std_logic;
         signal clr_bclk : std_logic;
   signal datain : std_logic;

   signal shift_in : std_logic_vector ( 15 downto 0);
         signal shift_counter : integer := 15;

         -- Second clock divider

         signal lrck_div2 : std_logic_vector (11 downto 0);
         --signal set_lrck2 : std_logic;
         signal bclk_divider2: std_logic_vector (7 downto 0);

begin
   -- LRCK divider
   -- Audio chip main clock is 18.432MHz / Sample rate 48KHz
```

```
-- Divider is 18.432 MHz / 48KHz = 192 (X"C0")
-- Left justify mode set by I2C controller

process(clk, reset_n) -- loops Another divider to slow down the LRclk
begin
   if ( reset_n = '0' ) then
      lrck_div2 <= (others => '0');
   elsif ( clk'event and clk='1' ) then
      if ( lrck_div2 = X"47F")  then     -- 8FF = 900 - 1
         lrck_div2 <= X"000";
      else
         lrck_div2 <= lrck_div2 + '1';
      end if;
   end if;
end process;

process(clk, reset_n) -- loops second bclk_divider -- we only need one of the 2
begin
   if ( reset_n = '0' ) then
      bclk_divider2 <= (others => '0');
   elsif ( clk'event and clk='1' ) then
      if ( bclk_divider2 = X"47" or set_lrck = '1')  then    -- 8F = 90-1
         bclk_divider2 <= X"00";
      else
         bclk_divider2 <= bclk_divider2 + '1';
      end if;
   end if;
end process;

process ( lrck_div2 )
begin
   if ( lrck_div2 = X"47F") then
      set_lrck <= '1';
   else
      set_lrck <= '0';
   end if;
end process;

-- Here we just have to change set_lrck to set_lrck2 to change the Sampling rate to 8kHz

process ( clk, reset_n)
begin
   if ( reset_n = '0') then
      lrck <= '0';
   elsif ( clk 'event and clk = '1') then
      if ( set_lrck = '1') then
         lrck <= not lrck;
      end if;
   end if;
end process;

-- BCLK divider
process ( bclk_divider2 )
begin
   if ( bclk_divider2  = X"23") then -- x5 -- why 5 and B?
      set_bclk <= '1';
```

```
      else
         set_bclk <= '0';
      end if;

      if ( bclk_divider2 = X"47") then -- xB
         clr_bclk <= '1';
      else
         clr_bclk <= '0';
      end if;
   end process;

   process ( clk, reset_n)
   begin
      if ( reset_n = '0') then
         bclk <= '0';
      elsif ( clk 'event and clk = '1') then
         if ( set_lrck = '1' or clr_bclk = '1') then
            bclk <= '0';
         elsif ( set_bclk = '1') then
            bclk <= '1';
         end if;
      end if;
   end process;

         process (clk)
         begin
         if ( clk 'event and clk = '1') then
                  if (set_bclk = '1') then
                           shift_in(shift_counter) <= AUD_ADCDAT;
                           if (shift_counter = 0) then
                                    shift_counter <= 15;
                           else
                                    shift_counter <= shift_counter - 1;
                           end if;
                  end if;
         end if;
         end process;

         process(clk)
   begin
      if ( clk'event and clk='1' ) then -- why??
         lrck_lat <= lrck;
      end if;
   end process;

--          process (clk)
--    begin
--       if ( clk'event and clk = '1') then
--          --if (( lrck_lat = '1' and lrck = '0') or ( lrck_lat = '0' and lrck = '1')) then
--          if(set_lrck <= '1') then
--                                    audio_req <= '1';
--          else
--             audio_req <= '0';
--          end if;
--       end if;
--    end process;
```

```
   -- Audio data shift output
   process ( clk, reset_n)
   begin
      if ( clk 'event and clk = '1') then
         if ( set_lrck = '1') then
            data_out <= shift_in;
                                          audio_req <= '1';
                                 else
                                          audio_req <='0';

         end if;
      end if;
   end process;

   -- Audio outputs

   AUD_BCLK     <= bclk;
           AUD_ADCLRCK              <= lrck;

end architecture;
```

5.3 de2_vga_raster.vhd

VGA Controller – Adapted from Lab 3

```
--------------------------------------------------------------------------------
--
-- Simple VGA raster display
--
-- Stephen A. Edwards
-- sedwards@cs.columbia.edu
--
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.CONV_STD_LOGIC_VECTOR;

entity de2_vga_raster is

  port (
    reset : in std_logic;
    clk   : in std_logic;                  -- Should be 25.125 MHz

            read : in std_logic;
            write : in std_logic;
            chipselect : in std_logic;
            address : in unsigned(4 downto 0);
            readdata : out unsigned(15 downto 0);
            writedata : in unsigned(15 downto 0);

    VGA_CLK,                      -- Clock
    VGA_HS,                      -- H_SYNC
    VGA_VS,                      -- V_SYNC
    VGA_BLANK,                       -- BLANK
    VGA_SYNC : out std_logic;        -- SYNC
    VGA_R,                       -- Red[9:0]
    VGA_G,                       -- Green[9:0]
```

```
    VGA_B : out unsigned(9 downto 0) -- Blue[9:0]
    );

end de2_vga_raster;

architecture rtl of de2_vga_raster is

  -- Video parameters

  constant HTOTAL       : integer := 800;
  constant HSYNC        : integer := 96;
  constant HBACK_PORCH  : integer := 48;
  constant HACTIVE      : integer := 640;
  constant HFRONT_PORCH : integer := 16;

  constant VTOTAL       : integer := 525;
  constant VSYNC        : integer := 2;
  constant VBACK_PORCH  : integer := 33;
  constant VACTIVE      : integer := 480;
  constant VFRONT_PORCH : integer := 10;

  constant RECTANGLE_HSTART : integer := 100;
  constant RECTANGLE_HEND   : integer := 540;
  constant RECTANGLE_VSTART : integer := 100;
  constant RECTANGLE_VEND   : integer := 380;



  -- Signals for the video controller
  signal Hcount : unsigned(12 downto 0);  -- Horizontal position (0-800)
  signal Vcount : unsigned(12 downto 0);  -- Vertical position (0-524)
  signal EndOfLine, EndOfField : std_logic;

  signal vga_hblank, vga_hsync,
    vga_vblank, vga_vsync : std_logic;  -- Sync. signals



--Signals written to Avalon Bus

  --Paddle Y position
  signal paddleY: unsigned(12 downto 0) := "0000100100010";
  --X and Y coordinates at the center of the ball
  signal XCO : unsigned(12 downto 0) := "0000110010000";
  signal YCO : unsigned(12 downto 0) := "0000100100010";

  --Scores
  signal myScore: unsigned(3 downto 0) := "0000";
  signal compScore: unsigned(3 downto 0) := "0000";

  --Denote whether blocks have been destroyed
  signal compBlock1: std_logic := '1';
  signal compBlock2: std_logic := '1';
  signal compBlock3: std_logic := '1';
  signal compBlock4: std_logic := '1';
  signal compBlock5: std_logic := '1';
```

```
 signal compBlock6: std_logic := '1';

 --Signals to track charging and deployment of bullets
 signal bulletInFlight: std_logic;
 signal bulletCharged: unsigned(1 downto 0):= "00";
 signal bulletX: unsigned(12 downto 0):= "0000000000000";
 signal bulletY: unsigned(12 downto 0):= "0000000000000";

 --logic signals to determine how to paint a pixel
 signal paintRed : std_logic := '1';
 signal paintGreen: std_logic := '1';
 signal paintCompBlock: std_logic;
 signal paintBullet: std_logic;
 signal paintPaddle : std_logic;
 signal Cir : std_logic;


 --Used to determine if a pixel is inside the circle radius
 signal Xd : unsigned(12 downto 0);
 signal Yd : unsigned(12 downto 0);


--Constants
constant RADI : unsigned(12 downto 0) := "0000000001000";
constant PADDLEX: unsigned(12 downto 0) := "0000010100000";
constant BULLET_WIDTH: unsigned(12 downto 0) := "0000000001010";
constant BULLET_HEAD_LENGTH: unsigned(12 downto 0) := "0000000000101";

begin

 GetCoords : process (clk)
 begin
        if rising_edge(clk) then
          if (write = '1') AND (chipselect = '1') then
                if address = "00001" then
                        paddleY <= writedata(12 downto 0);
                elsif address = "00010" then
                  XCO <= writedata( 12 downto 0);
            elsif address = "00000" then
                  YCO <= writedata( 12 downto 0);
                elsif address = "00011" then
                  myScore <= writedata( 3 downto 0);
            elsif address = "00100" then
                  compScore <= writedata( 3 downto 0);

                elsif address = "00101" then
                  compBlock1 <= writedata(0);
                elsif address = "00110" then
                  compBlock2 <= writedata(0);
                elsif address = "00111" then
                  compBlock3 <= writedata(0);
                elsif address = "01000" then
                  compBlock4 <= writedata(0);
                elsif address = "01001" then
                  compBlock5 <= writedata(0);
                elsif address = "01010" then
```

```
                    compBlock6 <= writedata(0);

                  elsif address = "01011" then
                    bulletInFlight <= writedata(0);
                  elsif address = "01100" then
                    bulletCharged <= writedata(1 downto 0);
                  elsif address = "01101" then
                    bulletX <= writedata(12 downto 0);
                  elsif address = "01110" then
                    bulletY <= writedata(12 downto 0);


              end if;
            end if;
  end if;
end process GetCoords;

SpitCoords : process (clk)
begin
          if rising_edge(clk) then
            if (read = '1') and (chipselect = '1') then
                    if address = "00001" then
                      readdata(15 downto 13) <= "000";
                      readdata(12 downto 0) <=  paddleY ;
                    elsif address = "00010" then
                      readdata(15 downto 13) <= "000";
                      readdata(12 downto 0) <=  XCO ;
              elsif address = "00000" then
                      readdata(15 downto 13) <= "000";
                      readdata(12 downto 0) <=  YCO ;
                    else
                              readdata(15 downto 0) <= "0000000000000000";
              end if;
            end if;
 end if;
end process SpitCoords;

-- Horizontal and vertical counters

HCounter : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      Hcount <= (others => '0');
    elsif EndOfLine = '1' then
      Hcount <= (others => '0');
    else
      Hcount <= Hcount + 1;
    end if;
  end if;
end process HCounter;

EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

VCounter: process (clk)
begin
```

```
  if rising_edge(clk) then
    if reset = '1' then
      Vcount <= (others => '0');
    elsif EndOfLine = '1' then
      if EndOfField = '1' then
        Vcount <= (others => '0');
      else
        Vcount <= Vcount + 1;
      end if;
    end if;
  end if;
end process VCounter;

EndOfField <= '1' when Vcount(9 downto 0) = VTOTAL - 1 else '0';

-- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK

HSyncGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' or EndOfLine = '1' then
      vga_hsync <= '1';
    elsif Hcount(9 downto 0) = HSYNC - 1 then
      vga_hsync <= '0';
    end if;
  end if;
end process HSyncGen;

HBlankGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      vga_hblank <= '1';
    elsif Hcount(9 downto 0) = HSYNC + HBACK_PORCH then
      vga_hblank <= '0';
    elsif Hcount(9 downto 0) = HSYNC + HBACK_PORCH + HACTIVE then
      vga_hblank <= '1';
    end if;
  end if;
end process HBlankGen;

VSyncGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      vga_vsync <= '1';
    elsif EndOfLine ='1' then
      if EndOfField = '1' then
        vga_vsync <= '1';
      elsif Vcount(9 downto 0) = VSYNC - 1 then
        vga_vsync <= '0';
      end if;
    end if;
  end if;
end process VSyncGen;
```

```
VBlankGen : process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      vga_vblank <= '1';
    elsif EndOfLine = '1' then
      if Vcount(9 downto 0) = VSYNC + VBACK_PORCH - 1 then
        vga_vblank <= '0';
      elsif Vcount(9 downto 0) = VSYNC + VBACK_PORCH + VACTIVE - 1 then
        vga_vblank <= '1';
      end if;
    end if;
  end if;
end process VBlankGen;



VideoOut: process (clk, reset)
begin


            --Determine whether to paint ball
            if ( Hcount > XCO ) then
                        Xd <= Hcount - XCO;
            else
                        Xd <= XCO - Hcount;
            end if;
            if ( Vcount > YCO ) then
                        Yd <= Vcount - YCO;
            else
                        Yd <= YCO - Vcount;
            end if;

            if ( Xd <= RADI ) and ( Yd <= RADI ) then
              if (reset = '1') or ( Xd * Xd + Yd * Yd > RADI * RADI ) then
                        Cir <= '0';
              elsif ( Xd * Xd + Yd * Yd < RADI * RADI) then
                        Cir <= '1';
    end if;
    else
      Cir <= '0';
            end if;


          --Determine whether to paint paddle
          if (Vcount <= paddleY + 100) and (Vcount >= paddleY) then
                        if (Hcount >= PADDLEX) and (Hcount <= PADDLEX + 10) then
                                    paintPaddle <= '1';
                        else
                                    paintPaddle <= '0';
                        end if;
          else
                        paintPaddle <= '0';
          end if;
```

```
--Determine whether to paint a block
if(HCount >= 750) and (HCount < 800) then
	if(Vcount > 38) and (Vcount <=116) and (compBlock1 = '1') then
		paintCompBlock <= '1';
	elsif(Vcount > 117) and (Vcount <=195) and (compBlock2 = '1') then
		paintCompBlock <= '1';
	elsif(Vcount > 196) and (Vcount <=274) and (compBlock3 = '1') then
		paintCompBlock <= '1';
	elsif(Vcount > 275) and (Vcount <=353) and (compBlock4 = '1') then
		paintCompBlock <= '1';
	elsif(Vcount > 354) and (Vcount <=432) and (compBlock5 = '1') then
		paintCompBlock <= '1';
	elsif(Vcount > 433) and (Vcount <=511) and (compBlock6 = '1') then
		paintCompBlock <= '1';
	else
		paintCompBlock <='0';
	end if;
else
	paintCompBlock <= '0';
end if;

--Determine whether to paint a score square (green if score reached)
if (Vcount <= 50) and (Vcount > 40) then
	if(Hcount >= 200) and (Hcount <210) then
		if (myScore <= 0) then
			paintRed <= '1';
			paintGreen <= '0';
		else
			paintRed <= '0';
			paintGreen <= '1';
		end if;

	elsif(Hcount >= 220) and (Hcount <230) then
		if (myScore <= 1) then
			paintRed <= '1';
			paintGreen <= '0';
		else
			paintRed <= '0';
			paintGreen <= '1';
		end if;

	elsif(Hcount >= 240) and (Hcount <250) then
		if (myScore <= 2) then
			paintRed <= '1';
			paintGreen <= '0';
		else
			paintRed <= '0';
			paintGreen <= '1';
		end if;

	elsif(Hcount >= 260) and (Hcount <270) then
		if (myScore <= 3) then
			paintRed <= '1';
			paintGreen <= '0';
		else
			paintRed <= '0';
```

```
                paintGreen <= '1';
        end if;

elsif(Hcount >= 280) and (Hcount <290) then
        if (myScore <= 4) then
                paintRed <= '1';
                paintGreen <= '0';
        else
                paintRed <= '0';
                paintGreen <= '1';
        end if;



elsif(Hcount >= 600) and (Hcount <610) then
        if (compScore <= 0) then
                paintRed <= '1';
                paintGreen <= '0';
        else
                paintRed <= '0';
                paintGreen <= '1';
        end if;

elsif(Hcount >= 620) and (Hcount <630) then
        if (compScore <= 1) then
                paintRed <= '1';
                paintGreen <= '0';
        else
                paintRed <= '0';
                paintGreen <= '1';
        end if;

elsif(Hcount >= 640) and (Hcount <650) then
        if (compScore <= 2) then
                paintRed <= '1';
                paintGreen <= '0';
        else
                paintRed <= '0';
                paintGreen <= '1';
        end if;

elsif(Hcount >= 660) and (Hcount <670) then
        if (compScore <= 3) then
                paintRed <= '1';
                paintGreen <= '0';
        else
                paintRed <= '0';
                paintGreen <= '1';
        end if;

elsif(Hcount >= 680) and (Hcount <690) then
        if (compScore <= 4) then
                paintRed <= '1';
                paintGreen <= '0';
        else
                paintRed <= '0';
```

```
                                        paintGreen <= '1';
                                end if;

                        else
                                paintRed <= '0';
                                paintGreen <= '0';
                        end if;
                else
                        paintRed <= '0';
                        paintGreen <= '0';
                end if;


                --Determines whether to paint a bullet (charging or in flight)
                if (bulletInFlight = '1') or (bulletCharged > 0) then
                        if (Vcount >= bulletY) and (Vcount < bulletY + BULLET_WIDTH) then
                                if (Hcount >=bulletX) and (HCount < bulletX + BULLET_HEAD_LENGTH)
then
                                        paintBullet <= '1';
                                elsif (Hcount >=bulletX) and (HCount < bulletX + BULLET_WIDTH) then
                                        if (bulletCharged > 1) or (bulletInFlight = '1') then
                                                paintBullet <= '1';
                                        else
                                                paintBullet <= '0';
                                        end if;
                                elsif (bulletCharged = "11") or bulletInFlight = '1' then
                                        if (Hcount >=bulletX) and (HCount < bulletX +
BULLET_WIDTH+BULLET_HEAD_LENGTH) then
                                                if(Vcount - bulletY >= HCount - bulletX - BULLET_WIDTH)
and (Vcount - bulletY <= BULLET_HEAD_LENGTH) then
                                                        paintBullet <= '1';
                                                elsif(bulletY + BULLET_WIDTH - Vcount >= HCount -
bulletX - BULLET_WIDTH) and (Vcount - bulletY >= BULLET_HEAD_LENGTH) then
                                                        paintBullet <= '1';
                                                else
                                                        paintBullet <= '0';
                                                end if;
                                        else
                                                paintBullet <= '0';
                                        end if;
                                else
                                        paintBullet <= '0';
                                end if;
                        else
                                paintBullet <= '0';
                        end if;
                else
                        paintBullet <= '0';
                end if;

                --Choose color palette based on logic signals
           if reset = '1' then
             VGA_R <= "0000000000";
             VGA_G <= "0000000000";
             VGA_B <= "0000000000";
           elsif clk'event and clk = '1' then
```

```
    if Cir = '1' then
      VGA_R <= "1111111111";
      VGA_G <= "1111111111";
      VGA_B <= "1111111111";
            elsif paintBullet = '1' then
      VGA_R <= "1111111111";
      VGA_G <= "0010011011";
      VGA_B <= "0000000000";
            elsif paintPaddle = '1' then
                  VGA_R <= "1010101010";
      VGA_G <= "1010101010";
      VGA_B <= "1010101010";
           elsif paintRed = '1' then
      VGA_R <= "1111111111";
      VGA_G <= "0000000000";
      VGA_B <= "0000000000";
    elsif paintGreen = '1' then
      VGA_R <= "0000000000";
      VGA_G <= "1111111111";
      VGA_B <= "0000000000";
            elsif paintCompBlock = '1' then
      VGA_R <= "1010101010";
      VGA_G <= "1010101010";
      VGA_B <= "1010101010";

    elsif vga_hblank = '0' and vga_vblank ='0' then
      VGA_R <= "0000000000";
      VGA_G <= "0000000000";
      VGA_B <= "0000000000";

             else
      VGA_R <= "0000000000";
      VGA_G <= "0000000000";
      VGA_B <= "0000000000";
    end if;
  end if;
 end process VideoOut;

 VGA_CLK <= clk;
 VGA_HS <= not vga_hsync;
 VGA_VS <= not vga_vsync;
 VGA_SYNC <= '0';
 VGA_BLANK <= not (vga_hsync or vga_vsync);

end rtl;
```

5.4 de2_audio_top.vhd
Audio Controller passes sound byte to Avalon Bus

```
--
-- DE2 top-level module that includes the simple audio component
--
-- Stephen A. Edwards, Columbia University, sedwards@cs.columbia.edu
--
-- From an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
```

```
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity de2_audio_top is

  port (
    -- Clocks

     CLOCK_50,                          -- 50 MHz


            --Bus components

            read : in std_logic;
            write : in std_logic;
            chipselect : in std_logic;
            address : in unsigned(4 downto 0);
            readdata : out unsigned(15 downto 0);
            writedata : in unsigned(15 downto 0);


            audio_data_in : in std_logic_vector(15 downto 0);
            audio_request : in std_logic

);
end de2_audio_top;

architecture datapath of de2_audio_top is




begin
  SpitPitch : process (CLOCK_50)
  begin
            if rising_edge(CLOCK_50) then
              if (read = '1') and (chipselect = '1')   then
                        readdata(15 downto 0) <= "0000000000000000";
                        if (audio_data_in(0) = '1') then
                                  readdata(0) <= '1';
                        end if;
                        if (audio_data_in(1) = '1') then
                                  readdata(1) <= '1';
                        end if;
                        if (audio_data_in(2) = '1') then
                                  readdata(2) <= '1';
                        end if;
                        if (audio_data_in(3) = '1') then
                                  readdata(3) <= '1';
                        end if;
                        if (audio_data_in(4) = '1') then
                                  readdata(4) <= '1';
                        end if;
```

```
                    if (audio_data_in(5) = '1') then
                            readdata(5) <= '1';
                    end if;
                    if (audio_data_in(6) = '1') then
                            readdata(6) <= '1';
                    end if;
                    if (audio_data_in(7) = '1') then
                            readdata(7) <= '1';
                    end if;
                    if (audio_data_in(8) = '1') then
                            readdata(8) <= '1';
                    end if;
                    if (audio_data_in(9) = '1') then
                            readdata(9) <= '1';
                    end if;
                    if (audio_data_in(10) = '1') then
                            readdata(10) <= '1';
                    end if;
                    if (audio_data_in(11) = '1') then
                            readdata(11) <= '1';
                    end if;
                    if (audio_data_in(12) = '1') then
                            readdata(12) <= '1';
                    end if;
                    if (audio_data_in(13) = '1') then
                            readdata(13) <= '1';
                    end if;
                    if (audio_data_in(14) = '1') then
                            readdata(14) <= '1';
                    end if;
                    if (audio_data_in(15) = '1') then
                            readdata(15) <= '1';
                    end if;
            end if;
  end if;
 end process SpitPitch;


end datapath;
```

5.6 pongbat.c

C file – software for Pongbat game

```
/*Software to control pongbat game logic and components
 * Written in 2009 by:
 * Peter Capraro
 * Michael Hankin
 * Anand Rajeswaran
 */

#include <io.h>
#include <system.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define RADIUS 8 //ball radius
```

```
#define PI 3.14159265
#define PADDLE_X 160
#define PADDLE_WIDTH 10
#define BULLET_WIDTH 10

void chooseGame();
void getFrequency(int current_sample);
int collision(double ball_x_center, double ball_y_center, int paddle_top, int paddle_height);

//Variables for frequency algorithm
int *periods;
int on = 0;                    //if on = 1, signal is hi, we are counting consectutive hi's
int sound = 0;                   //if 0, there is no sound coming in (only noise)
int silences = 0;                //number of consecutive "silent" samples
int consec_hi = 0;                //number of consecutive hi samples
int consec_lo = 0;                //number of consecutive lo samples
int replaceIndex = 0;                 //Index of periods array in which to place current index
int output = 176;                  //Sum of periods array
int avgNum=50;                     //Number of period measurements to store at once
//Overall Game controls
int gameStarted=0;           //Turns to 1 when paddle control option is chosen
int myScore=0;               //User score
int compScore=0;              //"Computer" score
int option =1;               //1 if frequency control, 0 if paddle control

//Bullet control variables
int bulletInFlight=0;        //Bullet has been fired
int bulletX;                 //Bullet X Coordinate
int bulletY;                 //Bullet Y Coordinate
int bulletCharged=0;         //Increments from 0 to 3 while bullet is charged

//Paddle control
int paddleDirection=0;       //0 if not moving, 1 if moving down, -1 if up
int paddle_top=200;          //Y coordinate of top of paddle
int paddle_height=100;       //length of paddle
//Ball Control
int angle_of_motion=30;      //Direction (degrees)
double ball_x_center=180;    //X Coordinate
double ball_y_center=90;     //Y Coordinate
int speed = 5;               //Magnitude of speed

//Delay variables
int loops_per_repaint=500;  //Number of iterations between execution of ball/paddle/bullet control
int count = 0;               //Count variable to track iterations

int main()
{
    int i;
    int compBlock[] = {1,1,1,1,1,1}; //Keep track of which blocks are destroyed
    periods= (int*)malloc(sizeof(int)*avgNum); //Array of the last avgNum results of the freq. algorithm
    //Initialized to 8 (to start as an average frequency)
    for (i = 0 ; i < avgNum ; i++)
    {
        periods[i]=8;
    }

    //Clean raster registers
    for (i = 0 ; i < 32 ; i++)
```

```
{
   IOWR_16DIRECT(RASTER_BASE, i, 0x0100);
}


chooseGame();
int maxOut=0;
int minOut=1000;
for(;;)
{
    //Get sound byte from audio controller
    int soundByte = IORD_16DIRECT(AUDIOSLAVE_BASE,0);

    //Call frequency algorithm
    getFrequency(soundByte);

    //Implement delay (don't execute every iteration)
    if ( count++ == loops_per_repaint )
    {

        count = 0;
    }
    if ( count %loops_per_repaint == 1 )
    {
        if(option) //Frequency controlled paddle
        {
                if(!sound) //Silence, don't move
                    paddleDirection = 0;
                else if(output/avgNum < 20) //Low periods, move up
                   paddleDirection = 1;
               else if(output/avgNum > 30) //High periods, move down
                  paddleDirection = -1;

                if(paddleDirection == 1 && paddle_top > 38) //Paddle not already at top
                    paddle_top -= 5;
                else if(paddleDirection == -1 && paddle_top < 412) //paddle not already at bottom
                    paddle_top += 5;
        }
        else //Noise controlled paddle
            if(sound) //Noise detected
            {

                if(paddle_top < 412)
                    paddle_top +=3;
            }
            else //Silence detected
            {
                if(paddle_top > 38)
                    paddle_top -=2;
            }


        if(bulletInFlight) //Bullet has been fired
        {
            if(bulletX + 15 >=750) //Bullet reaches X level of blocks
            {
                //If a bullet hits the blocks at a boundary, it can destroy two blocks
                int blockIndex1 = (bulletY - 38)/79;
```

```
        int blockIndex2 = (bulletY + BULLET_WIDTH - 38)/79;

        compBlock[blockIndex1] = 0;
        compBlock[blockIndex2] = 0;
        bulletInFlight = 0;
    }
    else
        bulletX+=4;
}
else //Bullet possibly charging
{
    //keep bullet aligned with center of paddle as paddle moves
    bulletY = paddle_top + paddle_height/2;
    bulletX = PADDLE_X;
}



//Ball collides with top of screen
if ( ball_y_center - RADIUS < 38 )
{//Reflect direction
    angle_of_motion = -angle_of_motion;
    ball_y_center = 39 +RADIUS;
}
//Ball collides with bottom of screen
if ( ball_y_center + RADIUS > 512 )
{
    angle_of_motion = -angle_of_motion;
    ball_y_center = 511 - RADIUS;
}
//Ball moves left past the paddle
if ( ball_x_center - RADIUS < 170 - RADIUS )
{
    //Computer scores
    compScore++;
    //Reset blocks, bullet, ball
    bulletCharged = 0;
    bulletInFlight = 0;
    ball_x_center=180;
    ball_y_center=90;
    angle_of_motion = 30;
    for(i=0;i<6;i++)
            compBlock[i] = 1;
}
//Ball reaches blocks on right
if( ball_x_center + RADIUS > 750 )
{
    //Check which blocks it is making contact with
    int blockIndex1 = (ball_y_center+RADIUS - 38)/79;
    int blockIndex2 = (ball_y_center-RADIUS - 38)/79;

    //if blocks have not been destroyed, reflect ball
    if(compBlock[blockIndex1] || compBlock[blockIndex2])
        angle_of_motion = -angle_of_motion + 180;
    else //Ball found a gap between blocks
    {
        //User scores, reset ball, bullets and blocks
        myScore++;
```

```
            bulletCharged = 0;
            bulletInFlight = 0;
            ball_x_center=180;
            ball_y_center=90;
            angle_of_motion = 30;
            for(i=0;i<6;i++)
                compBlock[i] = 1;
        }
    }
    //Check if ball in contact with paddle
    int collision_redirect = collision(ball_x_center, ball_y_center, paddle_top, paddle_height);

    if(collision_redirect != -1) //Ball is in contact with paddle
    {
        if(bulletCharged < 3) //Continue charging bullet
            bulletCharged++;
        else //Bullet fully charged, fire bullet
        {
            bulletInFlight = 1;
            bulletCharged = 0;
        }
        //Deflect ball at angle calculated by collision method (paddle gradient)
        angle_of_motion = collision_redirect;
    }
    angle_of_motion = angle_of_motion % 360;

    //Update ball position using trigonometry
    ball_x_center = ball_x_center + speed*cos(PI*angle_of_motion/180);
    ball_y_center = ball_y_center + speed*sin(PI*angle_of_motion/180);

    //Rewrite registers
    IOWR_16DIRECT(RASTER_BASE, 0x0004, (int)ball_x_center );
    IOWR_16DIRECT(RASTER_BASE, 0x0000, (int)ball_y_center );
    IOWR_16DIRECT(RASTER_BASE, 0x0002,paddle_top);
    IOWR_16DIRECT(RASTER_BASE, 0x0006,myScore);
    IOWR_16DIRECT(RASTER_BASE, 0x0008,compScore);
    IOWR_16DIRECT(RASTER_BASE, 0x000A,compBlock[0]);
    IOWR_16DIRECT(RASTER_BASE, 0x000C,compBlock[1]);
    IOWR_16DIRECT(RASTER_BASE, 0x000E,compBlock[2]);
    IOWR_16DIRECT(RASTER_BASE, 0x0010,compBlock[3]);
    IOWR_16DIRECT(RASTER_BASE, 0x0012,compBlock[4]);
    IOWR_16DIRECT(RASTER_BASE, 0x0014,compBlock[5]);
    IOWR_16DIRECT(RASTER_BASE, 0x0016,bulletInFlight);
    IOWR_16DIRECT(RASTER_BASE, 0x0018,bulletCharged);
    IOWR_16DIRECT(RASTER_BASE, 0x001A,bulletX);
    IOWR_16DIRECT(RASTER_BASE, 0x001C,bulletY);

    //Game is over
    if(myScore==5 || compScore==5)
    {
        //reset score, choose new game
         myScore = 0;
         compScore = 0;
         gameStarted=0;
         paddle_top = 200;
        chooseGame();
    }
}
```

```
    }
    return 0;
}

int collision(double ball_x_center, double ball_y_center, int paddle_top, int paddle_height)
{//Returns the (positive) degree at which to project the ball
 //Returns -1 if no collision
    if(ball_x_center - RADIUS <= PADDLE_X+PADDLE_WIDTH) //Ball crosses vertical line of paddle
    {
        if( (ball_y_center + RADIUS >= paddle_top) && (ball_y_center -RADIUS <=paddle_top +
paddle_height)) //Ball in contact with paddle
        {//(ball_y_center - paddle_top)/paddle_height = (redirect_angle/140) - 70
            //Implements paddle gradient per above equation
            int redirect_angle;
            redirect_angle = (int)((ball_y_center - paddle_top)/paddle_height*140)-70;
            return redirect_angle;
        }
    }
    return -1;
}
void chooseGame()
{//Paddle moves based on frequency, if moved to top of screen, noise control chosen, else frequency
controlled
    while(!gameStarted)
    {
         int soundByte;
        //Read sound byte from the avalon bus
        soundByte = IORD_16DIRECT(AUDIOSLAVE_BASE,0);
        getFrequency(soundByte);

        if ( count++ == 1000 )
        {
            count = 0;
        }
        if ( count %1000 == 1 )//Only execute every 1000 iterations
        {
            //printf("%d\n",output);
            if(!sound) //Silence, don't move
                paddleDirection = 0;
            else if(output/avgNum < 20) //Low periods, move up
                paddleDirection = 1;
            else if(output/avgNum > 30) //High periods, move down
                paddleDirection = -1;

            if(paddleDirection == 1 && paddle_top > 38) //Paddle not already at top
                paddle_top -= 3;
            else if(paddleDirection == -1 && paddle_top < 412) //paddle not already at bottom
                paddle_top += 3;

            if(paddle_top < 45) //Paddle reaches top, choose noise control
            {
                option = 0;
                gameStarted = 1;
            }
            if(paddle_top > 400) //Paddle reaches bottom, choose frequency control
            {
                option = 1;
                gameStarted = 1;
```

```
            }
            IOWR_16DIRECT(RASTER_BASE, 0x0002,paddle_top); //Output paddle height to VGA
        }
    }
}
void getFrequency(int current_sample)
{//current_sample is a int representation of a wav sound btye

    if(current_sample < 100 || current_sample > 65436 || (current_sample> 32000 && current_sample <
33000) )       //any thing of smaller magnitude is background noise
    {
        silences++;
        if(silences > 50)  //50 consecutive silent notes
            sound = 0;
    }
    else
    {
        sound = 1;
        silences = 0;
    }
    if(sound) //Sound is being played
    {
        if(on) //Last sound byte was positive
        {
            if(current_sample < 32768) //Positive means first bit is 1
                consec_hi = consec_hi + 1;
            else //The end of a positive string
            {
                on = 0;
                //Next 3 lines place consecutive high bytes in periods array and update sum of the array
(output)
                if(consec_hi > 10)
                {
                    output += consec_hi;
                    //printf("high %d\n",consec_hi);
                    output -= periods[replaceIndex];
                    periods[replaceIndex] = consec_hi;

                    replaceIndex= (replaceIndex+1)%avgNum;
                }
                consec_hi = 0;
            }
        }
        else
        {//Analogous to preceding if block
            if(current_sample >= 32768)
                consec_lo=consec_lo+1;
            else
            {

                on=1;
                if(consec_lo > 10)
                {
                    output += consec_lo;
                    //printf("low %d\n",consec_lo);
                    output -= periods[replaceIndex];
                    periods[replaceIndex] = consec_lo;
                    replaceIndex= (replaceIndex+1)%avgNum;
```

```
                }
                consec_lo = 0;
            }

        }
    }

}
```