Final Report: Autotune

By Cam Coleman (cc4535), Adam Banees (ab4972), and Khaela Harrod (klh2173)

Table of Contents

Introduction

In this project, we will create a program that utilizes an MP3 player and microphone to correct the pitch of any given sound and play it through a set of headphones. These hardware functionalities will be used to test our software algorithm implemented throughout.

We were able to get audio data from a MP3 Player through the audio CODEC and store it into the FPGA memory. Once all devices and codes are configured, the audio data is then read through the device drivers via Avalon bus and altered through our autotune algorithm. Once altered, the audio file is then sent back through the bus and written into the FPGA memory and output the pitch shift through the speaker.

System Overview

MP3 Player and Microphone

Using Khaela's iPhone 13 Pro Max, we used a lightning jack to 3.5 mm headphone jack to connect to our De1-SoC board. With this part, we were then able to connect the headphone jack to a Y-part headphone jack converter. This converter allowed us to connect the cellular device and microphone. After compilation through Quartus, we can now play audio from any library on Khaela's device. Using a speaker provided by Adam, we used a Y-part converter to connect the speaker to the De1-SoC board. Though it appears as a speaker, it still has the functionalities of a microphone.

I2C and Audio CODEC

I2C has become a popular serial communication standard used in embedded systems to link microcontrollers and peripherals. The data line (SDA) and the clock line (SCL) are the two lines that I2C employs for communication. Communication takes place when one device pulls the data line down while the clock line is up on these high-voltage connections. The data transfer between the devices is then synchronized by switching the clock line. I2C permits the use of numerous devices, each with a distinct address, on a single bus. This makes it simple to interface with many peripherals, including sensors, EEPROMs, and LCD screens.

Programmed through I2C, pins from the audio chip are connected to the Qsys system. This system is designed to send audio data to the chip. For the audio source, we were able to modify the settings to ensure a high quality sound. Sample rates are used to reconstruct audio data during playback. For our design, we used a 96kHz sample rate. We obtained this sample rate by modifying the sequence in which SW3, SW4, and SW5 were configured. To get a sample rate of 96kHz, we placed SW3, SW4, SW5 down (0). All other switches were up (1).

System Block Diagram

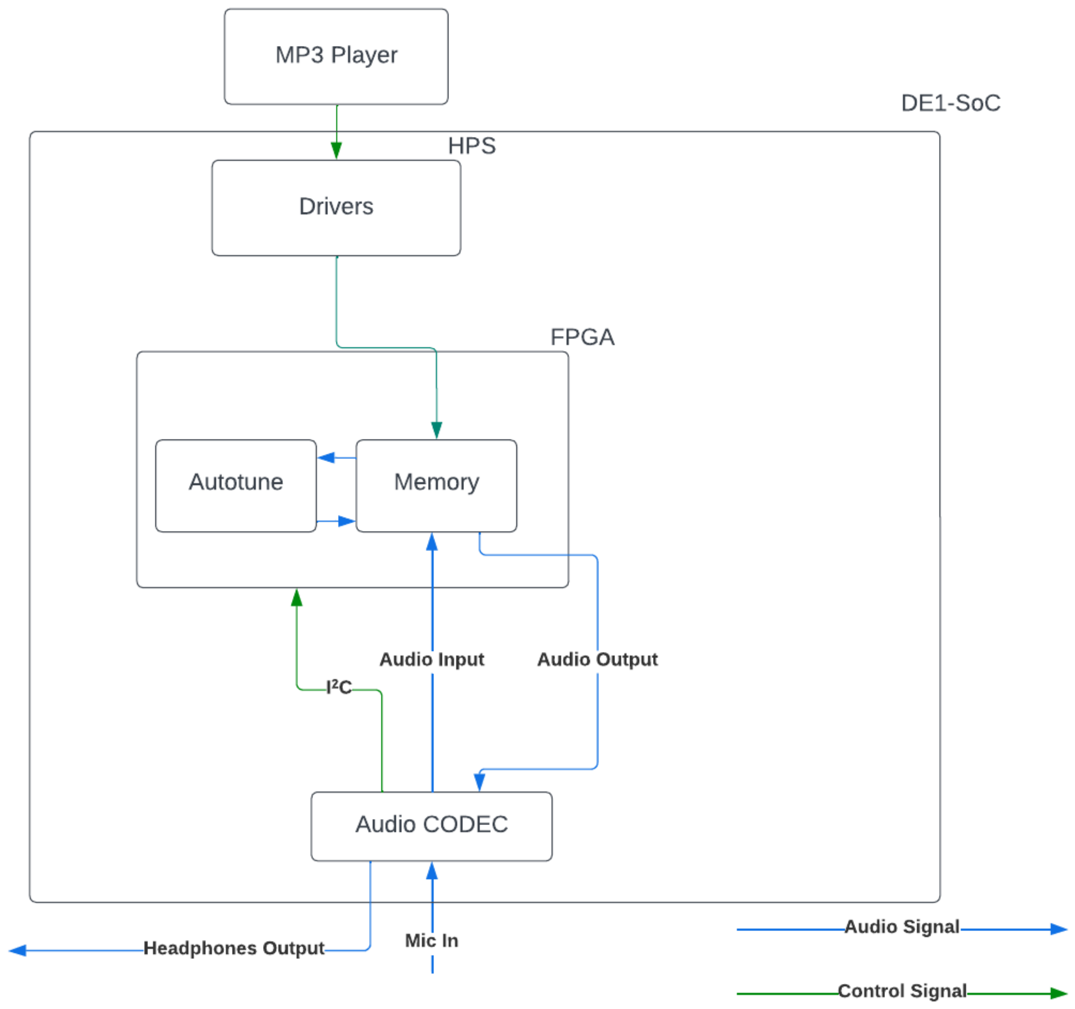


Figure 1: Block Diagram for Project Design

PitchShift Algorithm

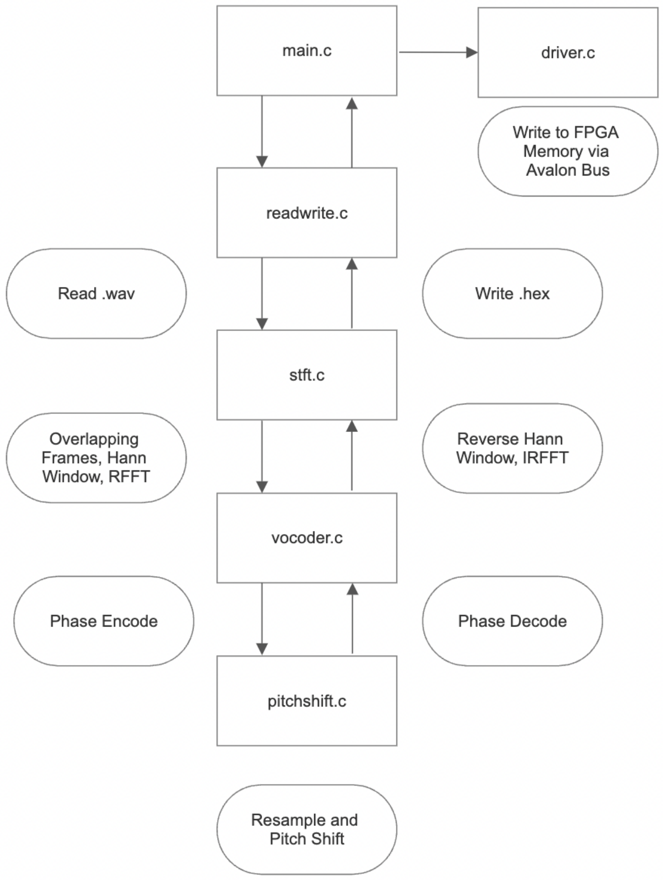


Figure 2: Attempted Pitchshift Algorithm

An attempt was made to create the pitch shift algorithm in C to work in software on our board. To optimize resource usage on the FPGA, the algorithm is implemented with fixed point operations. To handle this, an operations file was created. One other thing to note with the operations file is that it also handles complex operations with the real and imaginary part as a construct of 2 16-bit ints.

The first file of the algorithm is the readwrite. This simply reads and writes the data. We receive a .wav file and convert that to 16-bit PCM data. This also looks into the header information of the file to get important information like sample rate, bit width, and data format. The write would have converted the final processed signal into a .mif or .hex file to store in the registers of the FPGA using an Avalon Memory Bus.
The next file is the stft which handles both the Short Real Fast Fourier Transforms both forward and reverse. The FFT is used to analyze the signal in the frequency domain. The

algorithm that was attempted to be used for FFTs was Radix-2. Other methods that were used in this file were sliding_window_view, hann_window_forward, and hann_window_reverse. Sliding window view takes an array and creates a 2D array of bins with frames that are determined by the size and overlap. The Hann window is a commonly used window function for DSP applications which as follows:

$$w[n] = 0.5 - 0.5cos(2pi * n/N)$$

Next part of the algorithm is the phase vocoder which changes the time scale of the signal without changing the frequency content. There are two main functions of this algorithm which are encode and decode to handle the time stretching of the signal. Encode computes the frequency spectra and manipulates the phase information to achieve the time stretching. The decode process involves using the modified phase information and pitch shift and combining it with its original magnitudes. One last important thing about the vocoder is the phase wrapping which is under the wrap function to handle phase inconsistency.

The last file is the pitchshift file that takes in a factor to shift the pitch and apply it to the encoded signal. First it separates the encoded signal to magnitudes (real values) and frequencies (imaginary values) and then resamples it by linear interpolation with the linear function. Then the magnitudes are filtered by any frequencies that are less than 0 (DC component) and greater than half the sampling rate. (Nyquist frequency) Finally, the index of the frequency with the max magnitude is taken for each frame and reconstructs each frame with that magnitude and frequency.

Original implementation of the algorithm was attempted initially in hardware. There is an Intel FFT IP core that would have helped with the RFFT and IRFFT that we can write to with an Avalon Bus. It would have been nice to implement if it was not for the one hour time limit after the core instantiation and then we would have to buy the license.

The second implementation would have been using the KissFFT library that is known to be a simple implementation for 16-bit fixed point operations. This originally worked for regular FFTs but when we attempted to use the RFFT, it did not work properly. Finally we tried to implement a radix-2 algorithm but it did not work properly.

Conversion Software File

An additional attempt was made using software. The main idea was to use C programming to take a .wav file generated onto the board and be able to allow the board to produce the sound of the MIDI file at an altered pitch through the speakers as output. This step was more important for taking steps towards our second Milestone, which was to focus more on output. Unfortunately, we did not find a way to properly implement this algorithm. The algorithm is also still in its testing phase, where we give the software an already embedded .wav file and lets the software edit that file.

The “alsa/asoundlib.h” file was important for providing the programming interface with the ability to interact with the ALSA library, which enables the access and control of audio devices as well as manage sound input/output. This was used for the majority of the software file.

Resource Budgets

Using the Avalon audio interface, we can change the sample rate and bit depth of the data that the Audio CODEC is sending to the FPGA. If we have a sample rate of 44.1 kHz, a bit depth of 16 bits from 1 channel and 2.3 seconds of audio we get the following:

Total Number of samples = (Sample rate) * (Duration) = 44100 Hz * 2.294s = 101160 samples

Number of bytes per sample = (Bit depth) / 8 bytes per sample = 16 / 8 = 2 bytes per sample

Total Memory Required = (Total number of samples) * (Number of bytes per sample) = 101,160 * 2 = 202,320 bytes = 202.32 KB(approx.)

The audio file fits our FPGA memory constraint of ~ 440 KB so we will not have to worry of running out of space or forgetting about space for overhead.

One issue that we were having was implementing that algorithm created a lot of arrays that were needed for the algorithm which will cause memory issues.

Hardware and Software Interface

The main hardware-software interface we will be using in our project is the Avalon bus. Avalon interfaces simplify system design by allowing you to easily connect components in Intel FPGA. [1] We will be interfacing with the audio data sent from the Audio CODEC to the FPGA memory of the DE1-SoC board and implementing our autotune algorithm in the userspace. Our device driver will read the data coming from this memory and write the data after the audio file has been altered. In this project, we will use the Avalon Memory Mapped Interface (Avalon - MM) to read and write the addresses given from these registers.

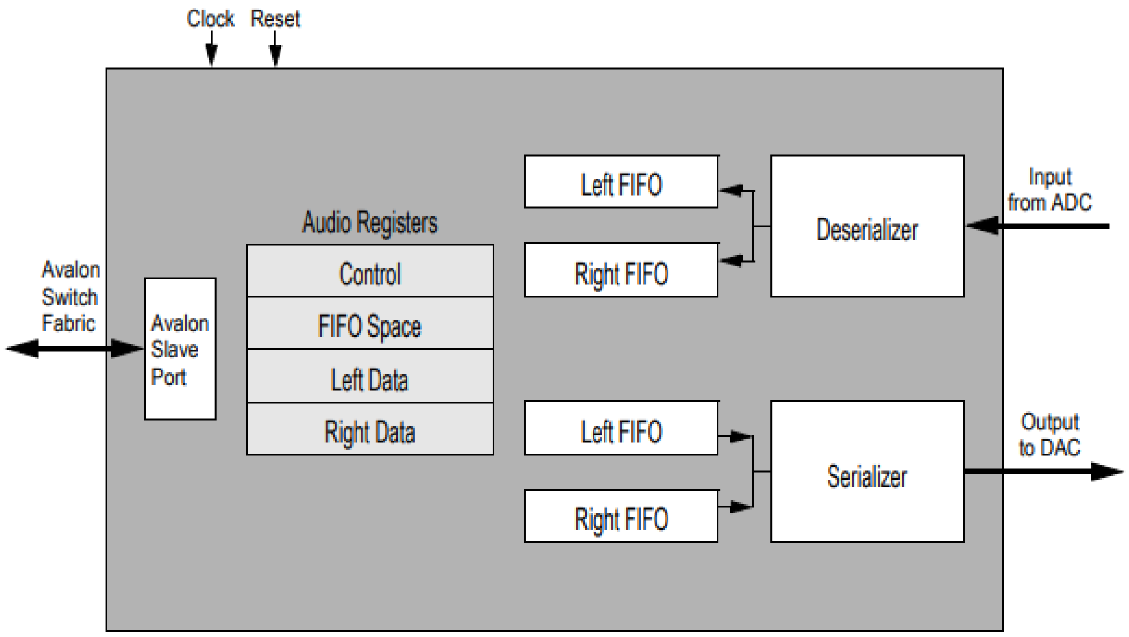


Figure 3: Block Diagram for Audio Core with Memory-Mapped Interface [2]

As for the interfacing for the keyboard, we will be using USB. We will implement something similar to Lab 2 with interfacing of the keyboard and simply program a record file to start recording our voice on the microphone by pressing spacebar.

Group Member Contributions

Khaela:

- Configuration of audio playback
- Configuration of audio CODEC
- Configuration of key switches

Adam:

- Core implementation of software pitchshift algorithm

Cameron:

- Research
- Debugging of Verilog files
- Configuration of MIDI to speaker
- Assistance with hardware algorithm and audio playback

References

[1]https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html
[2]https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/Audio_core.pdf
[3]https://learn.sparkfun.com/tutorials/i2c/all#introduction
[4]https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/fpga-to-hps-bridges-design-example.html
[5]https://www.projectrhea.org/rhea/index.php/Embedded_Fixed_Point_FFT
[6]https://www.juansaudio.com/amp/iir-vs-fir-understanding-their-differences
[7]http://www.cs.columbia.edu/~sedwards/classes/2019/4840-spring/reports/HotSprings.pdf
[8]http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf
[9]https://www.terasic.com.tw/en/
[10]https://github.com/evanfrazierc/FPGA-MP3-Player
[11]http://croakingkero.com/tutorials/load_wav/
[12]http://www.cs.columbia.edu/~sedwards/classes/2019/4840-spring/lab3.pdf
[13]https://stackoverflow.com/questions/16787117/load-wav-audio-file-in-memory-in-c
[14]https://github.com/jurihock/stftPitchShift

Project Files

I. Sound.v

```
// Modified by Khaela Harrod (klh2173)

module DE1_SoC_i2sound(

      output           ADC_CONVST,
      output           ADC_DIN,
      input            ADC_DOUT,
      output           ADC_SCLK,

      input            AUD_ADCDAT,
      inout            AUD_ADCLRCK,
      inout            AUD_BCLK,
      output           AUD_DACDAT,
      inout            AUD_DACLRCK,
      output           AUD_XCK,

      input            CLOCK2_50,
      input            CLOCK3_50,
      input            CLOCK4_50,
      input            CLOCK_50,
      output    [12:0] DRAM_ADDR,
      output    [1:0]  DRAM_BA,
      output           DRAM_CAS_N,
      output           DRAM_CKE,
      output           DRAM_CLK,
      output           DRAM_CS_N,
      inout     [15:0] DRAM_DQ,
      output           DRAM_LDQM,
      output           DRAM_RAS_N,
      output           DRAM_UDQM,
      output           DRAM_WE_N,
      output           FPGA_I2C_SCLK,
      inout            FPGA_I2C_SDAT,
      inout    [35:0]        GPIO_0,
      inout    [35:0]        GPIO_1,


      ///////// HEX0 /////////
      output     [6:0]  HEX0,

      ///////// HEX1 /////////
      output     [6:0]  HEX1,

      ///////// HEX2 /////////
      output     [6:0]  HEX2,

      ///////// HEX3 /////////
      output     [6:0]  HEX3,

      ///////// HEX4 /////////
      output     [6:0]  HEX4,

      ///////// HEX5 /////////
      output     [6:0]  HEX5,

`ifdef ENABLE_HPS
      ///////// HPS /////////
      inout            HPS_CONV_USB_N,
      output     [14:0] HPS_DDR3_ADDR,
      output     [2:0]  HPS_DDR3_BA,
      output           HPS_DDR3_CAS_N,
      output           HPS_DDR3_CKE,
      output           HPS_DDR3_CK_N,
      output           HPS_DDR3_CK_P,
      output           HPS_DDR3_CS_N,
      output     [3:0]  HPS_DDR3_DM,
      inout     [31:0] HPS_DDR3_DQ,
      inout     [3:0]  HPS_DDR3_DQS_N,
      inout     [3:0]  HPS_DDR3_DQS_P,
      output           HPS_DDR3_ODT,
      output           HPS_DDR3_RAS_N,
      output           HPS_DDR3_RESET_N,
      input            HPS_DDR3_RZQ,
      output           HPS_DDR3_WE_N,
      output           HPS_ENET_GTX_CLK,
      inout            HPS_ENET_INT_N,
      output           HPS_ENET_MDC,
      inout            HPS_ENET_MDIO,
      input          HPS_ENET_RX_CLK,
      input    [3:0] HPS_ENET_RX_DATA,
      input          HPS_ENET_RX_DV,
      output   [3:0] HPS_ENET_TX_DATA,
      output         HPS_ENET_TX_EN,
      inout    [3:0] HPS_FLASH_DATA,
      output         HPS_FLASH_DCLK,
      output         HPS_FLASH_NCSO,
      inout          HPS_GSENSOR_INT,
      inout          HPS_I2C1_SCLK,
      inout          HPS_I2C1_SDAT,
      inout          HPS_I2C2_SCLK,
      inout          HPS_I2C2_SDAT,
      inout          HPS_I2C_CONTROL,
      inout          HPS_KEY,
      inout          HPS_LED,
      inout          HPS_LTC_GPIO,
      output         HPS_SD_CLK,
      inout          HPS_SD_CMD,
      inout    [3:0] HPS_SD_DATA,
      output         HPS_SPIM_CLK,
      input          HPS_SPIM_MISO,
      output         HPS_SPIM_MOSI,
      inout          HPS_SPIM_SS,
      input          HPS_UART_RX,
      output         HPS_UART_TX,
      input          HPS_USB_CLKOUT,
      inout    [7:0] HPS_USB_DATA,
      input          HPS_USB_DIR,
      input          HPS_USB_NXT,
      output         HPS_USB_STP,
`endif /*ENABLE_HPS*/

      ///////// IRDA /////////
      input          IRDA_RXD,
      output         IRDA_TXD,

      ///////// KEY /////////
      input    [3:0] KEY,

      ///////// LEDR /////////
      output   [9:0] LEDR,

      ///////// PS2 /////////
      inout          PS2_CLK,
      inout          PS2_CLK2,
      inout          PS2_DAT,
      inout          PS2_DAT2,

      ///////// SW /////////
      input    [9:0] SW,

      ///////// TD /////////
      input          TD_CLK27,
      input    [7:0] TD_DATA,
      input           TD_HS,
      output          TD_RESET_N,
      input           TD_VS,

      ///////// VGA /////////
      output    [7:0] VGA_B,
      output          VGA_BLANK_N,
      output          VGA_CLK,
      output    [7:0] VGA_G,
      output          VGA_HS,
      output    [7:0] VGA_R,
      output          VGA_SYNC_N,
      output          VGA_VS
);
wire                        CLK_1M;
wire                        END;
wire                        KEYON;
wire      [23:0] AUD_I2C_DATA;
wire       GO;

assign              LEDR = 10'h000;
assign              AUD_DACDAT =AUD_ADCDAT ;

//I2C output data
CLOCK_500       u1(
                                    .CLOCK(CLOCK_50),
                                    .END(END),
                                    .RESET(KEYON),
                                    .CLOCK_500(CLK_1M),
                                    .GO(GO),
                                    .CLOCK_2(AUD_XCK),
                                    .DATA(AUD_I2C_DATA));

//i2c controller
i2c                         u2( // Host Side
                                    .CLOCK(CLK_1M),
                                    .RESET(1'b1),
                                     // I2C Side
                                    .I2C_SDAT(FPGA_I2C_SDAT),
                                    .I2C_DATA(AUD_I2C_DATA),
                                    .I2C_SCLK(FPGA_I2C_SCLK),
                                     // Control Signals
                                    .GO(GO),
                                    .END(END));

//KEY triggle
keytr                       u3(
                                    .clock(CLK_1M),
                                    .key(KEY[0]),
                                    .key1(KEY[1]),
                                    .KEYON(KEYON));


endmodule
```

II. i2c.v

```
module i2c (
  input         CLOCK,
  input         [23:0] I2C_DATA,
  input         GO,
  input         RESET,
  inout         I2C_SDAT,
  output        I2C_SCLK,
  output        END,
  output        ACK,
  output [5:0]  SD_COUNTER,
  output        SDO
);

  reg         SDO;
  reg         SCLK;
  reg         END;
  reg  [23:0]  SD;
  reg  [5:0]   SD_COUNTER;

  wire I2C_SCLK = SCLK | ((SD_COUNTER >= 4) & (SD_COUNTER <= 30)) ? ~CLOCK : 0;
  wire I2C_SDAT = SDO ? 1'bz : 0;

  reg  ACK1, ACK2, ACK3;
  wire ACK = ACK1 | ACK2 | ACK3;

  always @(negedge RESET or posedge CLOCK) begin
    if (!RESET) begin
      SD_COUNTER <= 6'b111111;
    end else begin
      if (GO == 0) begin
        SD_COUNTER <= 0;
      end else begin
        if (SD_COUNTER < 6'b111111) begin
          SD_COUNTER <= SD_COUNTER + 1;
        end
      end
    end
  end
```

```
always @(negedge RESET or posedge CLOCK) begin
 if (!RESET) begin
   SCLK  <= 1;
   SDO   <= 1;
   ACK1  <= 0;
   ACK2  <= 0;
   ACK3  <= 0;
   END   <= 1;
 end else begin
   case (SD_COUNTER)
     6'd0  : begin
       ACK1 <= 0;
       ACK2 <= 0;
       ACK3 <= 0;
       END  <= 0;
       SDO  <= 1;
       SCLK <= 1;
     end

     6'd1  : begin
       SD  <= I2C_DATA;
       SDO <= 0;
     end

     6'd2  : SCLK <= 0;
     6'd3  : SDO  <= SD[23];
     6'd4  : SDO  <= SD[22];
     6'd5  : SDO  <= SD[21];
     6'd6  : SDO  <= SD[20];
     6'd7  : SDO  <= SD[19];
     6'd8  : SDO  <= SD[18];
     6'd9  : SDO  <= SD[17];
     6'd10 : SDO  <= SD[16];
     6'd11 : SDO  <= 1'b1;

     6'd12 : begin
       SDO  <= SD[15];
       ACK1 <= I2C_SDAT;
     end
     6'd13 : SDO  <= SD[14];
     6'd14 : SDO  <= SD[13];
     6'd15 : SDO  <= SD[12];
     6'd16 : SDO  <= SD[11];
```

```
        6'd17 : SDO  <= SD[10];
        6'd18 : SDO  <= SD[9];
        6'd19 : SDO  <= SD[8];
        6'd20 : SDO  <= 1'b1;
        6'd21 : begin
          SDO  <= SD[7];
          ACK2 <= I2C_SDAT;
        end
        6'd22 : SDO <= SD[6];
        6'd23 : SDO <= SD[5];
        6'd24 : SDO <= SD[4];
        6'd25 : SDO <= SD[3];
        6'd26 : SDO <= SD[2];
        6'd27 : SDO <= SD[1];
        6'd28 : SDO <= SD[0];
        6'd29 : SDO <= 1'b1;

        6'd30 : begin
          SDO  <= 1'b0;
          SCLK <= 1'b0;
          ACK3 <= I2C_SDAT;
        end
        6'd31 : SCLK <= 1'b1;
        6'd32 : begin
          SDO <= 1'b1;
          END <= 1;
        end
      endcase
    end
  end

endmodule

module keytr (
        key,
        key1,
        ON,
        clock,
        KEYON,
        counter
       );
```

```
input           key;
input           key1;
input           clock;

output              ON;
output           KEYON;
output  [9:0]  counter;
reg     [9:0]  counter;

reg [3:0] sw;
reg flag,D1,D2;
reg [15:0] delay;
always@(negedge clock)
begin
  if (flag)
     sw<={key,sw[3:1]};
end
assign falling_edge = (sw==4'b0011)?1'b1:1'b0;

always@(posedge clock,negedge key1)
begin
  if (!key1)
     flag<=1'b1;
  else if (delay==15'd4096)
     flag<=1'b1;
  else if (falling_edge)
     flag<=1'b0;
end

always@(posedge clock)
begin
  if (!key)
     delay<=delay+1;
  else
     delay<=15'd0;
end

always@(negedge clock)
begin
  D1<=flag;
  D2<=D1;
end
assign KEYON = (D1 | !D2);
```

```
endmodule
```

III. conv.c

```
#include <stdio.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <alsa/asoundlib.h>

#define BUFF_SIZE 32768 //4096 * 8

typedef struct
{
  uint32_t samples;
  int16_t *data;
} sound_t;

sound_t hello;

//void adj_pitch(char *buffer, int br, float pitchFactor = )

int main(int argc, char *argv[])
{
  unsigned int sample_rate = 44100;
  int err;
  snd_pcm_t *handle;
  snd_pcm_hw_params_t *params;
  char *buffer;

  err = snd_pcm_open(&handle, "default", SND_PCM_STREAM_PLAYBACK, 0);
  if (err < 0)
    {
      printf("error opening PCM device \n");
      return 1;
    }

  err = snd_pcm_hw_params_malloc(&params);
  if (err < 0)
    {
      printf("error allocating hardware\n");
      return 1;
    }

  err = snd_pcm_hw_params_any(handle, paams);
  if (err < 0)
    {
      printf("error initializing hardware parameter structure\n");
      return 1;
    }

  if ((err = snd_pcm_hw_params_set_access(handle, params,
SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
    {
      printf("error setting access type\n");
      return 1;
    }

  if ((err = snd_pcm_hw_params_set_format(handle, params, SND_PCM_FORMAT_S16_LE)) < 0)
    {
      printf("error setting format \n");
      return 1;
    }

  if ((err = snd_pcm_hw_params_set_channels(handle, params, 2)) < 0)
    {
      printf("error with configuration space\n");
      return 1;
    }

  if ((err = snd_pcm_hw_params_set_rate_near(handle, params, &sample_rate,0)) < 0)
    {
      printf("error with empty configuration space\n");
      return 1;
    }

  if ((err = snd_pcm_hw_params(handle,params)) < 0)
    {
      printf("errr setting PCM hardware parameters\n");
      return 1;
    }
  buff = (char *)malloc(BUFF_SIZE * snd_pcm_format_width(SND_PCM_FORMAT_S16_LE) / 8 *
2);

  FILE *wavFile = fopen("./mono_voice_file.wav", “rb”);
  if (!wavFile)
    {
      print("error opening wav file\n");
      return 1;
    }

  snd_pcm_hw_params_free (hw_params);

  fprintf(stdout, "hw_params freed\n");

  vga_ball_arg_t vla;
  static const char filename[] = "/dev/vga_ball";

  if ( (vga_ball_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }


  while(!feof(wavFile)){

    /* allocate buffer for audio input */
    size_t nmemb = (BUFF_SIZE * snd_pcm_format_width(SND_PCM_FORMAT_S16_LE) / 8 * 2);
    int br = fread(buffer, 1, nmemb, wavFile);
    if (br < nmemb)
      {
       if((err = snd_pcm_writei(handle, buffer, br /
(snd_pcm_format_wiedth(SND_PCM_FORMAT_S16_LE) / 8 * 2))) == -EPIPE)
         {
           snd_pcm_prepare(handle);
           printf("underrun. prepared\n");
         }
       else
         {
           printf("error with received bytes\n");
           free(buffer);
           fclose(wavFile);
           snd_pcm_drain(handle);
```

```
            snd_pcm_close(handle);
            return 1;
          }
      }
      //incomp
      unsigned int pitchup = sample_rate * 2;
      snd_pcm_hw_params_set_rate_near(handle, params, &pitchup, 0);

  }
  //FREE THEM
  free(buffer);
  fclose(wavFile);
  snd_pcm_drain(handle);
  snd_pcm_close(handle);

  return 0;
}
```

IV. Clock.v

```
module CLOCK (
   CLOCK,
   CLOCK_500,
   DATA,
   END,
   RESET,
   GO,
   CLOCK_2
);

   input        CLOCK;
   input        END;
   input        RESET;

   output        CLOCK_500;
   output  [23:0] DATA;
   output        GO;
   output        CLOCK_2;

   reg   [10:0]  COUNTER_500;
   reg   [15:0]  ROM[7:0];
   reg   [15:0]  DATA_A;
   reg   [5:0]   address;

   wire  CLOCK_500 = COUNTER_500[9];
   wire  CLOCK_2 = COUNTER_500[1];
   wire  [23:0]  DATA = {8'h34, DATA_A};
   wire  GO = ((address <= 8'd8) && (END == 1)) ? COUNTER_500[10] : 1;

   always @(negedge RESET or posedge END) begin
      if (!RESET) begin
         address = 6'd0;
      end else if (address <= 6'd8) begin
         address = address + 1;
      end
   end

   reg [4:0] vol;
   wire [6:0] volume;
   always @(posedge RESET) begin
      vol = vol - 1;
   end
   assign volume = vol + 97;

   always @(posedge END) begin
      ROM[0] = 16'h0c00;
      ROM[1] = 16'h0ec2;
      ROM[2] = 16'h0838;

      ROM[3] = 16'h1000;

      ROM[4] = 16'h0017;
      ROM[5] = 16'h0217;
      ROM[6] = {8'h04, 1'b0, volume[6:0]};
      ROM[7] = {8'h06, 1'b0, volume[6:0]};

      ROM[8'd8] = 16'h1201;
      DATA_A = ROM[address];
   end

   always @(posedge CLOCK) begin
      COUNTER_500 = COUNTER_500 + 1;
   end
```

```
endmodule
```

V. Operations.h

```
#ifndef OPERATIONS_H
#define OPERATIONS_H

#include <stdint.h>

#define FRACTION_BITS 16

typedef int16_t fixed_point;
typedef struct {
   fixed_point real;
   fixed_point imag;
} complex_fixed;

// FIXED POINT OPERATIONS
fixed_point float_to_fixed(float x);
float fixed_to_float(fixed_point x);
fixed_point fixed_add(fixed_point x, fixed_point y);
fixed_point fixed_subtract(fixed_point x, fixed_point y);
fixed_point fixed_multiply(fixed_point x, fixed_point y);
fixed_point fixed_divide(fixed_point x, fixed_point y);

//COMPLEX FIXED POINT OPERATIONS
complex_fixed complex_float_to_fixed(float real, float imag);
complex_fixed complex_fixed_add(complex_fixed x, complex_fixed y);
complex_fixed complex_fixed_subtract(complex_fixed x, complex_fixed y);
complex_fixed complex_fixed_multiply(complex_fixed x, complex_fixed y);
complex_fixed complex_fixed_divide(complex_fixed x, complex_fixed y);

#endif
```

VI. Operations.c

```c
#include <stdint.h>
#include "operations.h"

#define FRACTION_BITS 8 // Define the number of fractional bits
#define PI_FIXED ((int16_t)(3.14159265358979323846 * (1 << 15)))

typedef int16_t fixed_point; // Define the fixed-point integer type

fixed_point float_to_fixed(float x) {
   return (int16_t)(x * (1 << FRACTION_BITS)); // Convert float to fixed-point
integer
}

float fixed_to_float(fixed_point x) {
   return (float)x / (1 << FRACTION_BITS); // Convert fixed-point integer to float
}

fixed_point fixed_add(fixed_point x, fixed_point y) {
   return x + y; // Addition of fixed-point integers
}

fixed_point fixed_subtract(fixed_point x, fixed_point y) {
   return x - y; // Subtraction of fixed-point integers
}

fixed_point fixed_multiply(fixed_point x, fixed_point y) {
   return (int16_t)(((int32_t)x * y) >> FRACTION_BITS); // Multiplication of
fixed-point integers
}

fixed_point fixed_divide(fixed_point x, fixed_point y) {
   return (int16_t)(((int32_t)x << FRACTION_BITS) / y); // Division of fixed-point
integers
}

complex_fixed complex_float_to_fixed(float real, float imag) {
   complex_fixed result;
   result.real = float_to_fixed(real);
   result.imag = float_to_fixed(imag);
   return result;
}

complex_fixed complex_fixed_add(complex_fixed x, complex_fixed y) {
   complex_fixed result;
   result.real = fixed_add(x.real, y.real);
   result.imag = fixed_add(x.imag, y.imag);
   return result;
}

complex_fixed complex_fixed_subtract(complex_fixed x, complex_fixed y) {
   complex_fixed result;
   result.real = fixed_subtract(x.real, y.real);
   result.imag = fixed_subtract(x.imag, y.imag);
   return result;
}

complex_fixed complex_fixed_multiply(complex_fixed x, complex_fixed y) {
   complex_fixed result;
   result.real = fixed_subtract(fixed_multiply(x.real, y.real),
fixed_multiply(x.imag, y.imag));
   result.imag = fixed_add(fixed_multiply(x.real, y.imag), fixed_multiply(x.imag,
y.real));
   return result;
}

complex_fixed complex_fixed_divide(complex_fixed x, complex_fixed y) {
   complex_fixed result;
   fixed_point denom = fixed_add(fixed_multiply(y.real, y.real),
fixed_multiply(y.imag, y.imag));
   result.real = fixed_divide(fixed_add(fixed_multiply(x.real, y.real),
fixed_multiply(x.imag, y.imag)), denom);
   result.imag = fixed_divide(fixed_subtract(fixed_multiply(x.imag, y.real),
fixed_multiply(x.real, y.imag)), denom);
   return result;
}

int16_t atan2_fixed(int16_t y, int16_t x) {
   int16_t quotient = 0;
   int16_t remainder = 0;

   if (x == 0) {
       if (y > 0) {
           return PI_FIXED/2;
       } else if (y < 0) {
           return -PI_FIXED/2;
       } else {
           return 0;
       }
   }

   quotient = (int16_t)(((int32_t)y << 15) / x);

   if (y == 0) {
       if (x > 0) {
           return 0;
       } else {
           return PI_FIXED;
       }
   } else if (y > 0) {
       if (x > 0) {
           remainder = (int16_t)(((int32_t)y << 15) - ((int32_t)x * quotient));
           return (int16_t)(atan_fixed(quotient, remainder) * (1 << 15));
       } else {
           remainder = (int16_t)(((int32_t)y << 15) - ((int32_t)x * quotient));
           return (int16_t)((atan_fixed(quotient, remainder) + PI_FIXED) * (1 <<
15));
       }
   } else {
       if (x > 0) {
           remainder = (int16_t)(((int32_t)y << 15) - ((int32_t)x * quotient));
           return (int16_t)(atan_fixed(quotient, remainder) * (1 << 15));
       } else {
           remainder = (int16_t)(((int32_t)y << 15) - ((int32_t)x * quotient));
           return (int16_t)((atan_fixed(quotient, remainder) - PI_FIXED) * (1 <<
15));
       }
   }
}
```

```c
int16_t atan_fixed(int16_t quotient, int16_t remainder) {
   int32_t x2 = (int32_t)quotient * quotient;
   int32_t a = x2 * 15708;     // 15708 = 4 * 65536 * tan(pi/4)
   int32_t b = ((int32_t)536870912 + a) >> 30; // 536870912 = 2^29
   int16_t c = (int16_t)(((int32_t)quotient << 14) / ((int32_t)536870912 - b));
   int16_t d = (int16_t)(2 * c / ((int32_t)1 + x2 * (int32_t)c * c / ((int32_t)1 <<
28)));
   return (int16_t)(((int32_t)d * (int32_t)remainder) / (1 << 15));
}

int16_t hypot_fixed(int16_t x, int16_t y) {
   double x_f = (double)x / 32767.0;
   double y_f = (double)y /32767.0;;
   double r = (x_f * x_f) + (y_f * y_f);
   double hypot_f = sqrt(r);
   int16_t hypot_i = (int16_t)round(hypot_f * 32767.0);
   return hypot_i;
}
```

VII. stft.h

```c
#ifndef STFT_H
#define STFT_H

#include <stdint.h>
#include "operations.h"

#define FRAME_SIZE 1024
#define HOP_SIZE 32
#define FFT_SIZE_IN_BITS 10

int16_t** sliding_window(int16_t* x, int num_samples, int framesize, int hopsize);
void apply_hann_window_forward(int16_t *input);
void apply_hann_window_reverse(int16_t *input);
void rfft_fixed(complex_fixed *x, int N, int inverse);

#endif
```

VIII. stft.c

```c
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <string.h>
#include "stft.h"
#include "operations.h"

#define PI 3.14159265358979323846
#define FRAME_SIZE 512
#define HOP_SIZE 32
#define FIXED_POINT_SHIFT 16
#define FFT_SIZE_IN_BITS 10
#define FRACT_BITS FIXED_POINT_SHIFT-1
#define FRACT_SCALE (1 << FRACT_BITS)

int16_t** sliding_window(int16_t* x, int num_samples, int framesize, int hopsize) {
   int num_frames = (num_samples - framesize) / hopsize + 1;
   int16_t** frames = malloc(num_frames * sizeof(int16_t*));

   for (int i = 0; i < num_frames; i++) {
       frames[i] = malloc(framesize * sizeof(int16_t));
       for (int j = 0; j < framesize; j++) {
           frames[i][j] = x[i * hopsize + j];
       }
   }

   return frames;
}

void apply_hann_window_forward(int16_t *input) {
   // Create a Hann window
   int16_t window[FRAME_SIZE];
   for (int i = 0; i < FRAME_SIZE; i++) {
       double value = 0.5 * (1.0 - cos(2.0 * PI * i / (FRAME_SIZE - 1)));
       window[i] = (int16_t)(value * pow(2, FIXED_POINT_SHIFT));
   }
```

```
    // Apply the Hann window to the input data
    for (int i = 0; i < FRAME_SIZE; i++) {
        input[i] = (input[i] * window[i]) >> FIXED_POINT_SHIFT;
    }
}

void apply_hann_window_reverse(int16_t *input) {
    // Create a Hann window
    int16_t window[FRAME_SIZE];
    for (int i = 0; i < FRAME_SIZE; i++) {
        double value = 0.5 * (1.0 - cos(2.0 * PI * i / (FRAME_SIZE - 1)));
        window[i] = (int16_t)(value * pow(2, FIXED_POINT_SHIFT));
    }

    // Apply the Hann window to the input data
    for (int i = 0; i < FRAME_SIZE; i++) {
        input[i] = (input[i] * window[i]) >> FIXED_POINT_SHIFT;
    }
}

// Radix-2 RFFT function
void rfft_fixed(complex_fixed *x, int N, int inverse) {
    // Bit-reversal
    int k, j, m;
    for (k = 1, j = 0; k < N; ++k) {
        int i;
        for (i = N >> 1; i > (j ^= i); i >>= 1);
        if (k < j) {
            complex_fixed temp = x[j];
            x[j] = x[k];
            x[k] = temp;
        }
    }

    // Cooley-Tukey algorithm
    int n = 2;
    int shift = 1;
    while (n <= N) {
        int mmax = n >> 1;
        int angle = 0;
        int m;
        for (m = 0; m < mmax; ++m) {
            int k;
            int16_t wr, wi;
            if (inverse) {
                wr = cos(angle * PI / (n >> 1)) * 32767.0;
                wi = sin(angle * PI / (n >> 1)) * -32767.0;
            } else {
                wr = cos(angle * PI / (n >> 1)) * 32767.0;
                wi = sin(angle * PI / (n >> 1)) * 32767.0;
            }
            angle += shift * 2 * mmax;

            for (k = m; k < N; k += n) {
                int16_t tr = wr * x[k + mmax].real - wi * x[k + mmax].imag;
                int16_t ti = wr * x[k + mmax].imag + wi * x[k + mmax].real;
                x[k + mmax].real = x[k].real - tr;
                x[k + mmax].imag = x[k].imag - ti;
                x[k].real += tr;
                x[k].imag += ti;
            }
        }
        n <<= 1;
        shift = inverse ? -shift : -shift;
    }

    if (inverse) {
        // Scale the output
        int i;
        for (i = 0; i < N; ++i) {
            x[i].real /= N;
            x[i].imag /= N;
        }
    }
}
```

IX. vocoder.h

```c
#ifndef VOCODER_H
#define VOCODER_H

#include <stdint.h>
#include "operations.h"

#define FRAME_SIZE 1024
#define HOP_SIZE 32
#define FFT_SIZE_IN_BITS 10

int16_t wrap(int16_t x);
void encode(complex_fixed *complex_data, int framesize, int hopsize, int samplerate,
complex_fixed *encoded_data);
void decode(complex_fixed *data, int framesize, int hopsize, int samplerate,
complex_fixed *decoded_data);

#endif
```

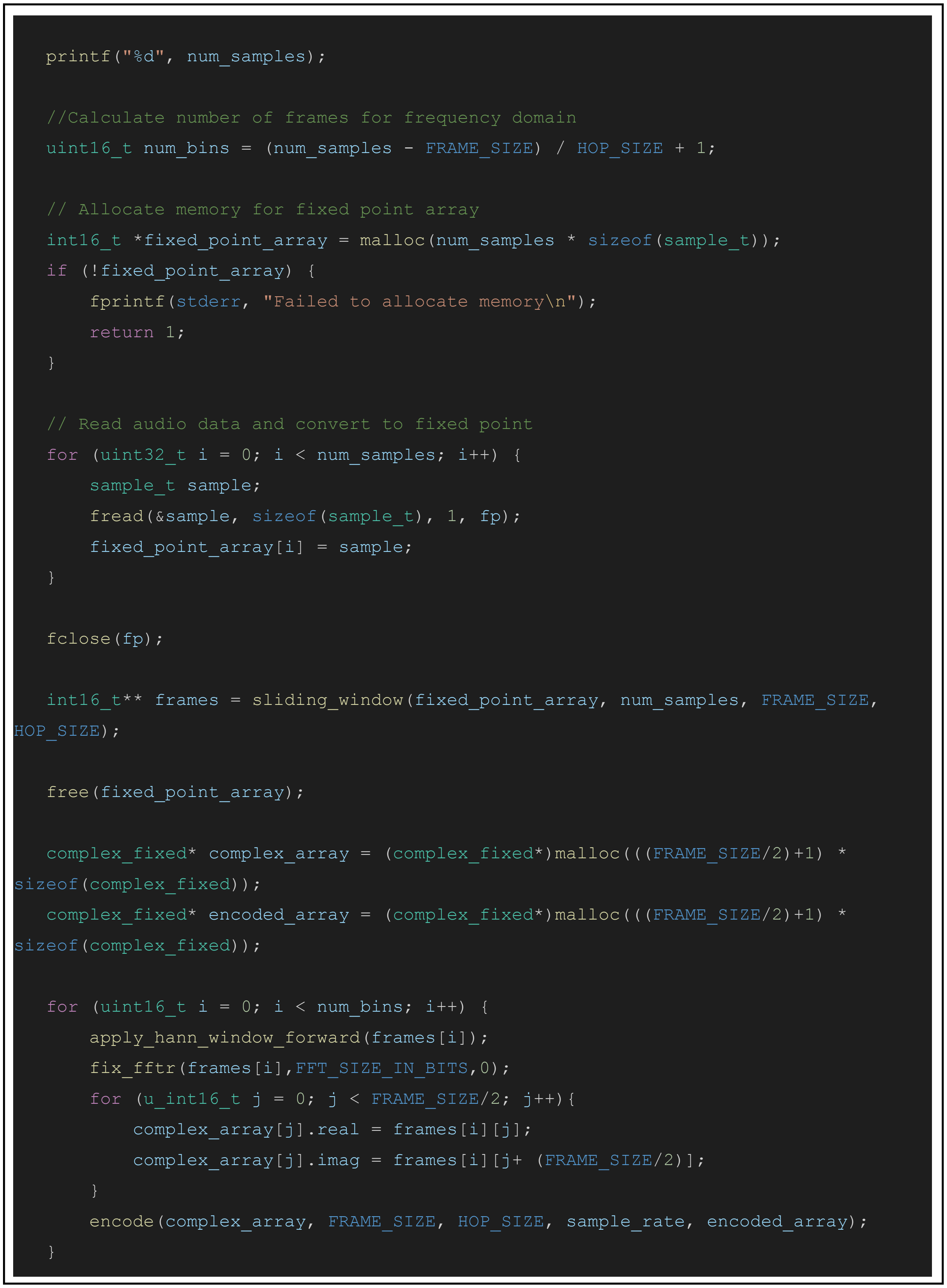
X. vocoder.c

```c
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <string.h>
#include "stft.h"
#include "operations.h"

#define PI_FIXED ((int16_t)(3.14159265358979323846 * (1 << 15)))

int16_t wrap(int16_t x) {
   const int16_t pi = 3217;     // 3.14159265358979323846 in Q15 format
   const int16_t pi2 = 6435;    // 2*pi in Q15 format

   x = (x + pi) % pi2;
   if (x < 0) {
       x += pi2;
   }
   return x - pi;
}

void encode(complex_fixed *complex_data, int framesize, int hopsize, int samplerate,
complex_fixed *encoded_data) {
   int N = framesize; // number of samples per frame
   int16_t phaseinc = 2.0 * PI_FIXED * hopsize / framesize; // phase increment
between adjacent samples
   int16_t freqinc = samplerate / framesize; // frequency increment
   complex_fixed buffer[N]; // to store previous frame
   complex_fixed frame[N];
   int16_t abs[N], arg[N];
   double delta[N], freq[N];
   for (int n = 0; n < N; n++) {
       abs[n] = hypot_fixed(complex_data[n].real, complex_data[n].imag);
       arg[n] = atan2_fixed(complex_data[n].imag, complex_data[n].real);
       delta[n] = arg[n] - buffer[n].imag;
       buffer[n].real = 0;
       buffer[n].imag = arg[n];
       freq[n] = wrap(delta[n] - n * phaseinc) / phaseinc;
       freq[n] = (n + freq[n]) * freqinc;
       encoded_data[n].real = (int16_t)round(abs[n]);
       encoded_data[n].imag = (int16_t)round(freq[n]);
   }
}

void decode(complex_fixed *data, int framesize, int hopsize, int samplerate,
complex_fixed *decoded_data) {
   int N = framesize; // number of samples per frame
   int16_t phaseinc = 2.0 * PI_FIXED * hopsize / framesize; // phase increment
between adjacent samples
   int16_t freqinc = samplerate / (double)framesize; // frequency increment
   complex_fixed buffer[N]; // to store previous frame
   double abs[N], freq[N], delta[N], arg[N];
   for (int n = 0; n < N; n++) {
       abs[n] = (double)data[n].real;
       freq[n] = (double)data[n].imag;
       delta[n] = (n + (freq[n] - n * freqinc) / freqinc) * phaseinc;
       buffer[n].real = round(abs[n] * cos(delta[n]));
       buffer[n].imag = round(abs[n] * sin(delta[n]));
       decoded_data[n].real = buffer[n].real;
       decoded_data[n].imag = buffer[n].imag;
   }
}
```

XI. Readwrite.c

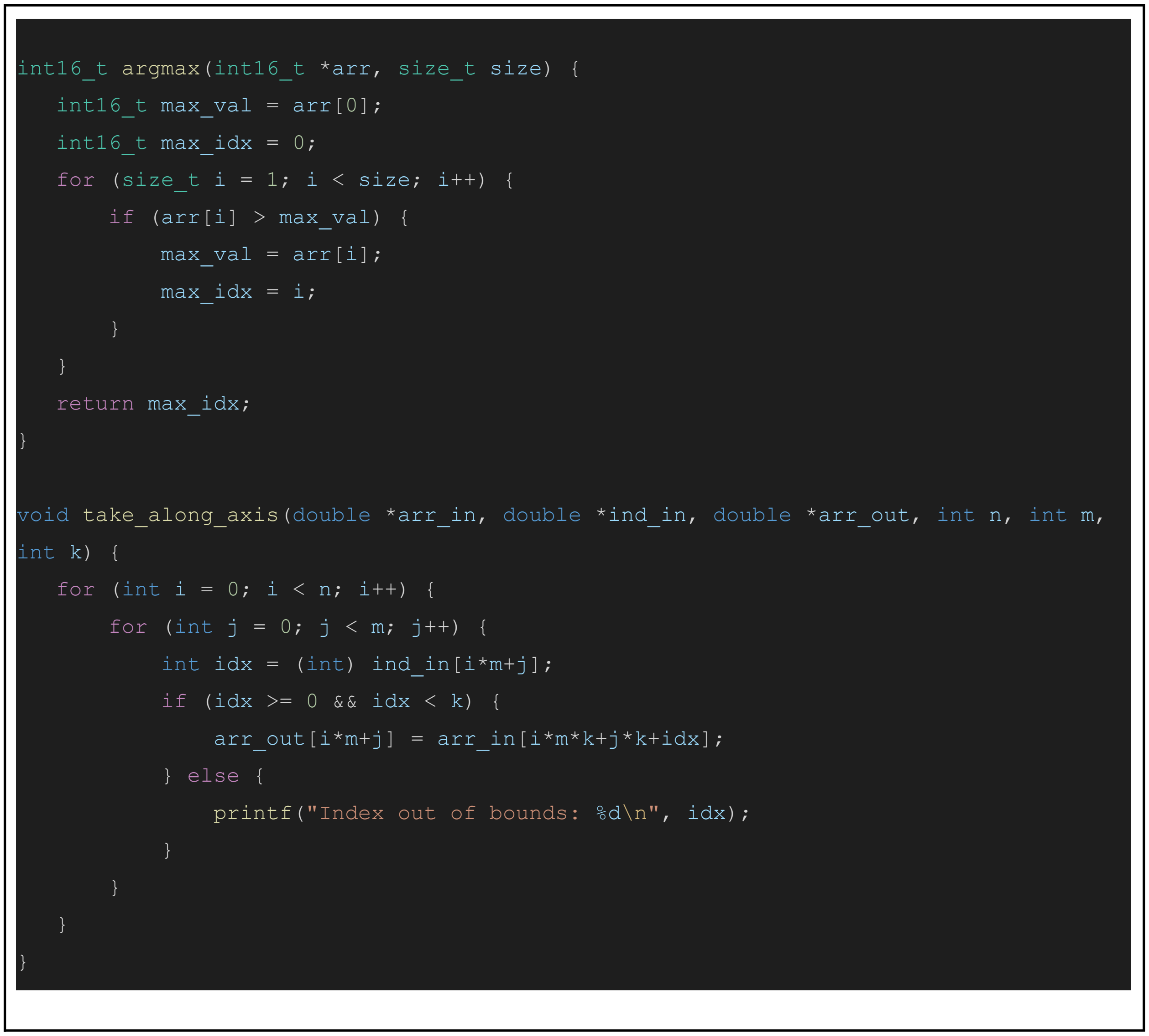
```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <string.h>
#include "operations.h"
#include "stft.h"
#include "vocoder.h"

#define PI_FIXED ((int16_t)(3.14159265358979323846 * (1 << 15)))
#define FRAME_SIZE 1024
#define HOP_SIZE 32
#define FIXED_POINT_SHIFT 16
#define FFT_SIZE_IN_BITS 10
#define FRACT_BITS FIXED_POINT_SHIFT-1
#define FRACT_SCALE (1 << FRACT_BITS)
#define FACTOR 2

typedef int16_t sample_t;

int16_t* linear(const int16_t* x, size_t n, int16_t factor, size_t* m);
int16_t argmax(int16_t *arr, size_t size);
void take_along_axis(double *arr_in, double *ind_in, double *arr_out, int n, int m,
int k);
void clip(int *arr, int size, int min_val, int max_val);

int main(int argc, char *argv[]) {
   if (argc < 2) {
       fprintf(stderr, "Usage: %s input_file.wav\n", argv[0]);
       return 1;
```

```c
}

const char *input_file = argv[1];
FILE *fp = fopen(input_file, "rb");
if (!fp) {
    fprintf(stderr, "Failed to open file: %s\n", input_file);
    return 1;
}

// Read WAV header
char riff[4];
uint32_t file_size;
char wave[4];
char fmt[4];
uint32_t fmt_size;
uint16_t audio_format;
uint16_t num_channels;
uint32_t sample_rate;
uint32_t byte_rate;
uint16_t block_align;
uint16_t bits_per_sample;
char data[4];
uint32_t data_size;

fread(riff, 1, 4, fp);
fread(&file_size, 4, 1, fp);
fread(wave, 1, 4, fp);
fread(fmt, 1, 4, fp);
fread(&fmt_size, 4, 1, fp);
fread(&audio_format, 2, 1, fp);
fread(&num_channels, 2, 1, fp);
fread(&sample_rate, 4, 1, fp);
fread(&byte_rate, 4, 1, fp);
fread(&block_align, 2, 1, fp);
fread(&bits_per_sample, 2, 1, fp);
fread(data, 1, 4, fp);
fread(&data_size, 4, 1, fp);

//CONSTANTS
// Calculate number of audio samples
uint32_t num_samples = data_size / block_align;
```



```
    printf("%d", num_samples);

    //Calculate number of frames for frequency domain
    uint16_t num_bins = (num_samples - FRAME_SIZE) / HOP_SIZE + 1;

    // Allocate memory for fixed point array
    int16_t *fixed_point_array = malloc(num_samples * sizeof(sample_t));
    if (!fixed_point_array) {
        fprintf(stderr, "Failed to allocate memory\n");
        return 1;
    }

    // Read audio data and convert to fixed point
    for (uint32_t i = 0; i < num_samples; i++) {
        sample_t sample;
        fread(&sample, sizeof(sample_t), 1, fp);
        fixed_point_array[i] = sample;
    }

    fclose(fp);

    int16_t** frames = sliding_window(fixed_point_array, num_samples, FRAME_SIZE,
HOP_SIZE);

    free(fixed_point_array);

    complex_fixed* complex_array = (complex_fixed*)malloc(((FRAME_SIZE/2)+1) *
sizeof(complex_fixed));
    complex_fixed* encoded_array = (complex_fixed*)malloc(((FRAME_SIZE/2)+1) *
sizeof(complex_fixed));

    for (uint16_t i = 0; i < num_bins; i++) {
        apply_hann_window_forward(frames[i]);
        fix_fftr(frames[i],FFT_SIZE_IN_BITS,0);
        for (u_int16_t j = 0; j < FRAME_SIZE/2; j++){
            complex_array[j].real = frames[i][j];
            complex_array[j].imag = frames[i][j+ (FRAME_SIZE/2)];
        }
        encode(complex_array, FRAME_SIZE, HOP_SIZE, sample_rate, encoded_array);
    }
```

```c
    free(frames);
    free(complex_array);
    free(encoded_array);

    return 0;
}

int16_t* linear(const int16_t* x, size_t n, int16_t factor, size_t* m) {
    if (factor == 1.0) {
        *m = n;
        int16_t* y = malloc(n * sizeof(int16_t));
        if (!y) {
            return NULL;
        }
        for (size_t i = 0; i < n; i++) {
            y[i] = x[i];
        }
        return y;
    }

    *m = (size_t) (n * factor);
    int16_t* y = malloc((*m) * sizeof(int16_t));
    if (!y) {
        return NULL;
    }

    float q = (float) n / (*m);
    for (size_t i = 0; i < fmin(n, *m); i++) {
        float k = i * q;
        size_t j = (size_t) floorf(k);
        k -= j;

        if (j < n - 1) {
            y[i] = (int16_t) roundf((1.0f - k) * x[j] + k * x[j+1]);
        } else {
            y[i] = x[j];
        }
    }
    return y;
}
```



```c
int16_t argmax(int16_t *arr, size_t size) {
   int16_t max_val = arr[0];
   int16_t max_idx = 0;
   for (size_t i = 1; i < size; i++) {
       if (arr[i] > max_val) {
           max_val = arr[i];
           max_idx = i;
       }
   }
   return max_idx;
}

void take_along_axis(double *arr_in, double *ind_in, double *arr_out, int n, int m,
int k) {
   for (int i = 0; i < n; i++) {
       for (int j = 0; j < m; j++) {
           int idx = (int) ind_in[i*m+j];
           if (idx >= 0 && idx < k) {
               arr_out[i*m+j] = arr_in[i*m*k+j*k+idx];
           } else {
               printf("Index out of bounds: %d\n", idx);
           }
       }
   }
}
```