Synth

A MIDI-Controlled Synthesizer/Vocoder

CSEE 4840: Embedded System Design
Prof. Stephen Edwards
Final Project Report

Warren Cheng
Rob Hendry
Ashwin Ramachandran

1. Purpose

The objective of our project was to design and build a MIDI-controlled synthesizer and vocoder. Essentially, a user would play a note (or several notes) on the MIDI controller, a keyboard, and be able to produce various synthesized tones, such as a sine-wave, a sawtooth wave, or combination of such waves. In addition, our device has the added functionality of receiving input from a microphone (called a "modulator") and "modulating" it with the MIDI-controlled synthesized tone (referred to as the "carrier"), thereby distorting the microphone input in an interesting manner. If one's voice is used as microphone input, the resulting sound is sometimes referred to as "robotic sounding".

2. System Architecture

Our system is built using an Altera Cyclone II FPGA, and comprised of three main modules: the MIDI receiver and decoder module, the carrier synthesis module, and the vocoding module. At the top level, this collection of modules is assembled through the use of the Avalon bus (figure 1).

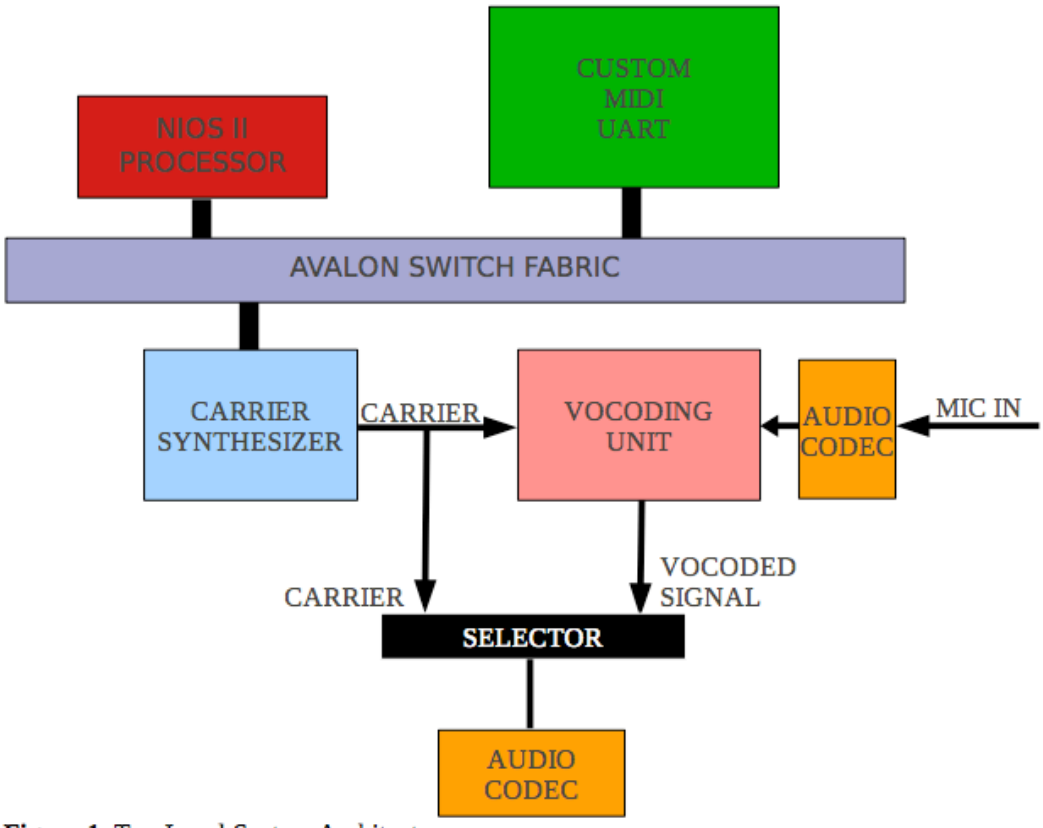


Figure 1. Top-Level System Architecture

The system thus consists of a Nios processor, which communicates with its slave peripherals, namely the UART and the carrier synthesizer, through the medium of the Avalon bus. Outside of this system is the vocoding unit (consisting of several filter banks) and the audio codec for interfacing with a speaker. Having described the general layout of the system, we proceed to discuss the design and implementation of each component in detail.

2.1 The MIDI Specification and UART

The Specification

MIDI is an asynchronous, serial communication specification that describes the transmission of musical events. According to the MIDI 1.0 specification, MIDI controllers send commands to MIDI devices in the form of messages. A message is composed of a series of bytes as depicted in figure 2.

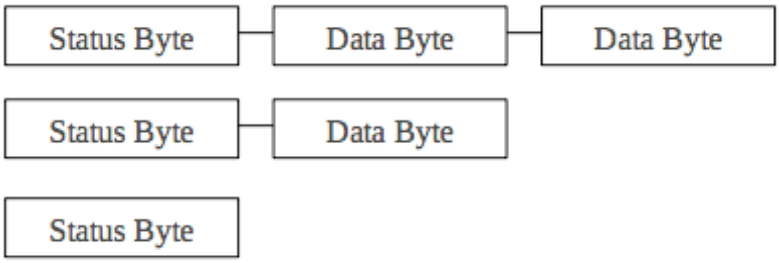


Figure 2. Sample Possible MIDI Message Formats

For this project, our interest lay in understanding and decoding only two types of messages: those that tell a device to play a note, and those that tell a device to stop playing a note. Each of these events is mapped to a special status signal in the MIDI specification – Note On and Note Off, respectively. The messages are constructed as follows.

Note On and Note Off events are marked by the transmission of a status signal. A status signal is a byte identified by a logic 1 in the MSB of the byte. This status byte can then be subdivided into two nibbles, the upper nibble indicating whether the controller is turning a note on or off, and the lower nibble

indicating which channel we are performing this operation on. Since our system consists of only one controller and one MIDI-sensitive device, we ignore the channel information for the sake of simplicity. A hexadecimal value of 9 (0x9) in the upper nibble indicates a Note On event, while a hexadecimal value of 8 (0x8) in the lower nibble indicates a Note Off event.

For a Note On event, the status byte is followed by two more bytes. The first byte is the note number. Because the MSB of this byte must be logic 0 (otherwise it would be a status signal), there are 128 possible notes that this byte can encode. The note number byte is followed by another byte – this is called the velocity byte. Velocity is a number that represents the volume of the note desired by the controller. This too can encode 128 different volume levels. Since our controller is not advanced enough to provide this many velocity values, we ignore velocity in our decoding. However, MIDI also defines a Note On event with velocity 0 to be equivalent to a Note Off event. Therefore, it is only in this case (where the velocity is 0) do we do something with the velocity information.

A Note Off event is followed by two bytes as well: the note and the velocity. The only difference here from the Note On event is the status reading 0x8.

The UART

Since MIDI is a bit-serial asynchronous protocol, the data bits that constitute a byte must be accumulated into a full byte, and then sent for processing. Such functionality is accomplished through the use of a UART (Universal Asynchronous Receiver/Transmitter).

MIDI controllers send bits out at a rate of 31250 kilobaud per second, or rather at a maximum rate of 31250 bits per second. Since this transmission is asynchronous, there is no clock to tell us when data is valid, or when we should expect a new bit. Therefore, a common sampling scheme (which is also

implemented in our UART) is oversampling the signal.

A MIDI byte is sent as follows: one start bit (logic 0), followed by the 8 data bits, ended by one stop bit (logic 1). Therefore, the UART begins operation by detecting the data line drop to logic 0 and then begins counting. Sampling the data at 16 samples per bit, off of a 50 MHz clock, for a 3125 kilobaud per second signal amounts to counting 100 clock cycles for each sample. For the start bit, the UART's counter counts 8 samples. Once it counts 8 samples, the UART is now looking at the middle of the bit. From here, it samples the signal 16 more times. The UART, now looking at the middle of the first bit, takes a sample. It counts another 16 samples to the middle of the 2^{nd} bit, and samples again. This process is repeated for the remaining bits. Finally, once all 8 bits have been read, the data byte is stored in an 8-bit register and a flag is raised to indicate that a new data byte is available for the processor. Upon reading the data itself, the processor clears the flag.

Sampling at the center of the bit has one important advantage: it provides the UART some robustness to variations from the nominal baud rate. Should the data be sampled at the edge of a bit, a small deviation in the baud rate could have a negative consequence on the data read by the UART.

Interfacing MIDI

Bringing the MIDI signals onto the board was quite interesting. The Altera DE-2 boards that we used have no MIDI-in port on them. Therefore we needed an alternative. The serial encoding of a MIDI byte is actually quite similar to that of the RS-232 serial protocol. Both have a start bit, 8 data bits, and a stop bit per byte of transmitted data. Even though the board has an RS-232 input, using it would require that we use a level shifter in our circuit due to RS-232's high voltage swings for encoding bit values.

Instead we settled on using the PS/2 port's data pin to bring our MIDI signals onto the board. PS/2 uses

reasonable voltage levels (3-5v) that would not require the use of a level shifter. The only piece of special equipment we needed was an optoisolator. This is because MIDI signals are encoded in current loops. Since there is no fixed ground for these loops, an optoisolator was required to change these current loop encodings into TTL values for the PS/2 data pin to accept. Our circuit was mildly adapted from that suggested by the MIDI Manufacturers Association themselves (figure 3).

In order to maintain a proper voltage reading at the input pins of the PS/2 port, we needed to drive the circuit with 5v power and ground from the PS/2 port's power and ground pins. This allowed for us to have a solid ground reference to which the PS/2 data pin would read ag st. The circuit is pictured in figure 4.

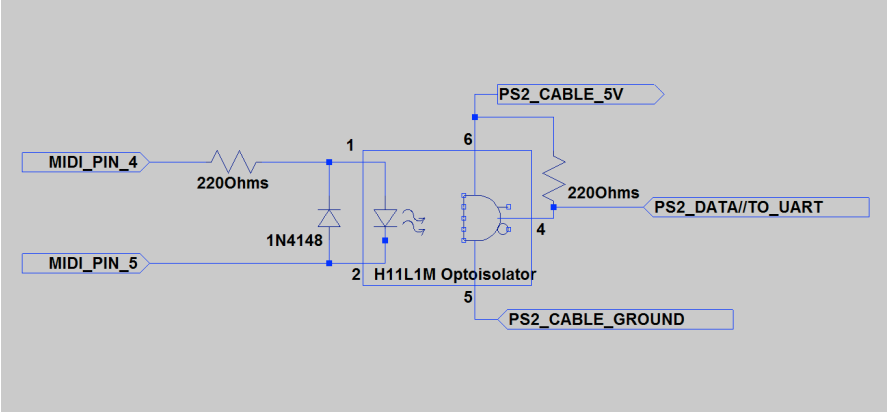


Figure 3. Circuit schematic for converting MIDI to PS/2 compatible TTL voltage

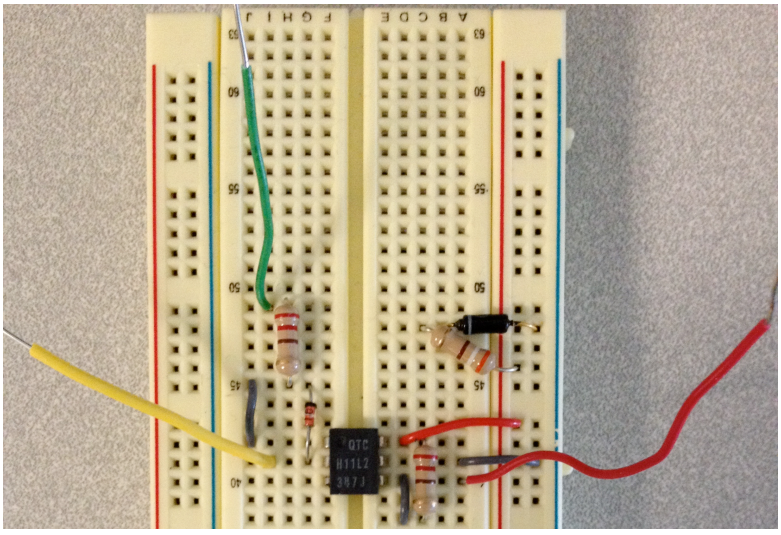


Figure 4. MIDI conversion circuit. Green and yellow flags indicate MIDI pins 5 and 4, respectively, while the red flag is the PS/2 data to the UART

2.2 The NIOS II and Interfacing

The Nios II was instantiated into the design using the SOPC builder and serves two purposes. First, the processor has the responsibility of polling the UART and, once a new data byte is available, decoding that data byte in the context of the MIDI specification and deciding on a course of action (whether to turn a note on or off). Second, it is the job of the processor to identify and keep track of the active notes being produced by the synthesizer, and appropriately writing the correct frequency inputs to the synthesizer logic. For example, if synthesizer two is playing the note A440, and the MIDI controller sends the system a note off command for A440, it is the responsibility of the processor to find out that synthesizer two is playing A440, and to silence it by writing the appropriate data value.

The Nios II processor is connected to the UART via parallel input/output ports to the Avalon bus. The data register and the status register both have their own dedicated PIO. After polling, once the processor sees that a new byte is ready, it clears the flag and reads the byte.

Additionally, the processor can write to any one of five synthesizer blocks in the FPGA. This interfacing is also done through PIO ports that connect to the inputs of the synthesizer blocks. In order to specify what note to produce, the processor simply writes the desired frequency in hexadecimal to the appropriate block.

2.3 The FM Synthesizer

The FM synthesizer VHDL component contains an audio codec driver and configurator (adapted from those given in lab three), and instantiates the five top-level carrier generating components, called FM patches. Each FM patch is a piece of hardware, which defines the way in which the five voice carriers are generated—typical synthesizer terminology uses the word “patch” to refer a particular configuration

of hardware which produces a particular sound or timbre. Following this definition, within the FM patch there are multiple FM operators. The FM operator generates a basic waveform given a pitch and modulates that carrier based on some modulator input.

Waveform Generation

One important aspect of implementing the synthesis hardware was generating waveforms which sounded appealing to the ear. A basic equation for generating a sine wave, which has a very "pure" sound to it, is: A = sin(wt), where "A" is the instantaneous amplitude of the output signal, "w" is the desire frequency (corresponding to the pitch, or note), and "t" is time. However, implementing a sine function in computer hardware is difficult and, more importantly, consumes a large amount of resources. A typical solution for VHDL programmers is to generate a simple look-up table that maps a discrete number of inputs to the sine function to the sine of those inputs. However, if the table is too small, the sine wave produces a raspy and distorted sound at the speaker. We found experimentally that a sine function which maps 512 inputs to their corresponding sine value over one complete oscillation of a sine wave produces a very pure tone. It is very hard to hear the difference in tone quality with tables that are larger than this.

Another aspect of generating sine waves was providing an appropriate external interface to the FM operator. In the implementation, the FM operator is given a precise frequency of the pitch to generate. However, waves are generated by stepping through the look-up table at a certain rate—changing the pitch of the generated wave is accomplished by taking more or less clock cycles to advance up one entry in the look-up table. Given the number of sample in the sine look-up table and a clock frequency, the FM operator converts actual frequencies into discrete numbers of clock cycles with the equation: (clock frequency)/(note frequency * number of look-up table samples). FM synthesis is accomplished by adding a modulating amplitude to that equation, making the output of the FM

operator:

*FMOut = sineTable[clock frequency / ((note frequency + modulation amplitude) * 512)]*

The FM operators also produce a saw-tooth wave, which yields a buzzing noise (but not too buzzy—the pitch is still discernible), and a waveform generated in a similar fashion to the sine wave except that all the values in the lookup table are random. The saw tooth wave is accomplished by simply counting from 0 to some peak amplitude over the course of a period of the note.

FM Patches

The challenge with creating FM patches is that it is hard to tell what the output of a modulated signal will sound like (as least to the relatively inexperienced synthesizer designer). A few attempts were made to produce interesting FM patches, all of which sounded pretty cool, but they corrupted the base frequency of the desired pitch so much that the resulting audible sound had very little relationship to the desired note. One alteration that can be made to the FM operator is a better interface for adjusting the amount of influence a modulated signal has on its carrier.

Interface with the MIDI system and Vocoder

The top-level module of the synthesizer provides a very simple interface to the MIDI controller: five frequency inputs with one for each voice, and five "enable" signals to turn each voice on and off. The vocoder aspect is intended to be instantiated within the synthesizer; the synthesizer provides a carrier signal and delivers the microphone input from the audio driver to the vocoder, and takes the output of the vocoder to the audio driver.

2.4 Vocoder

Background

A vocoder is a device which can generate a modulated version of a carrier signal. It was originally used to encrypt speech for telecommunications, but is now used popularly for musical applications. With the

right carrier input, you can distort your voice such that it sounds metallic or robotic. Basically the two signals that are required for the operation of a vocoder: a carrier signal which usually comes from a harmonically rich source such as an organ or synthesizer and a modulating signal which is typically the signal from a microphone.

The vocoder operates by decomposing human speech into a number of frequency bands. This is done by passing the voice signal through a series of band pass filters whose center frequencies cover the broad spectrum of human voice. These band pass filters have the ability to accentuate the frequencies of speech around the center frequency of the filter while suppressing others. These components of the voice signal are then passed through a square function, which are then passed through a low pass filter to get the average energy of that signal component at a certain point in time. At the same point in time, the carrier is passed through an identical bank of bandpass filters. The filtered voice signal output components are then multiplied by the corresponding filtered carrier outputs. Their products are finally summed and the result is the vocoded output.

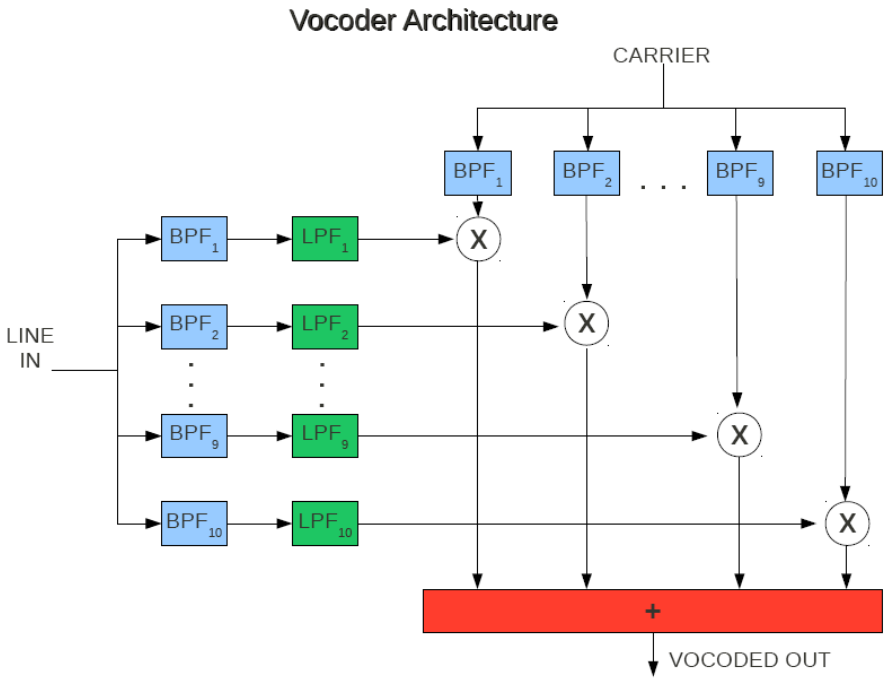


Figure 5. Block diagram of the Vocoder system.

Implementation

We originally wanted to process the signals through software using a series of FFTs, but after realizing the hardware limitations of the DE2 board, we realized that our output could not be produced in real-time. This is why we decided to go with a DSP approach.

DSP implementation of the channel vocoder basically entailed designing the filters outlined above in digital hardware. After discussions with the EE Department's Prof. Dan Ellis, we decided to design our filters using the following equations:

$$H_{BP}(z) = \frac{1-\alpha}{2}\,\frac{1-z^{-2}}{1-\beta(1+\alpha)z^{-1}+\alpha z^{-2}}$$

$$\omega_c = \cos^{-1}\beta$$

$$B = \cos^{-1}\left(\frac{2\alpha}{1+\alpha^2}\right)$$

Figure 6. Design equations used to build the bandpass filter.

$$H_{LP}(z) = K\,\frac{1+z^{-1}}{1-\alpha z^{-1}}$$

$$\alpha = \frac{1-\sin\omega_c}{\cos\omega_c}$$

Figure 7. Design equations used to build the lowpass filter.

For simulation purposes, MATLAB code (see Appendix C) was written in order to generate the bandpass and lowpass numerator and denominator coefficients when given the center/corner frequency, sampling frequency, and/or bandwidth. From these two functions, we were able to generate banks of filters and develop the vocoder system. Ultimately we determined that the best placement for the band pass filters for a pleasant sounding output was as follows:

Frequency	Bandwidth
111Hz	40Hz
250Hz	50Hz
354Hz	60Hz
500Hz	70Hz
707Hz	80Hz
1000Hz	90Hz
1414Hz	150Hz
2000Hz	250Hz
2828Hz	500Hz
5187Hz	1000Hz

In order for the recovered speech to be intelligible, the filters had to be spread out across audible spectrum and meet the condition that the bandwidths at lower frequencies were narrower than those at high frequencies.

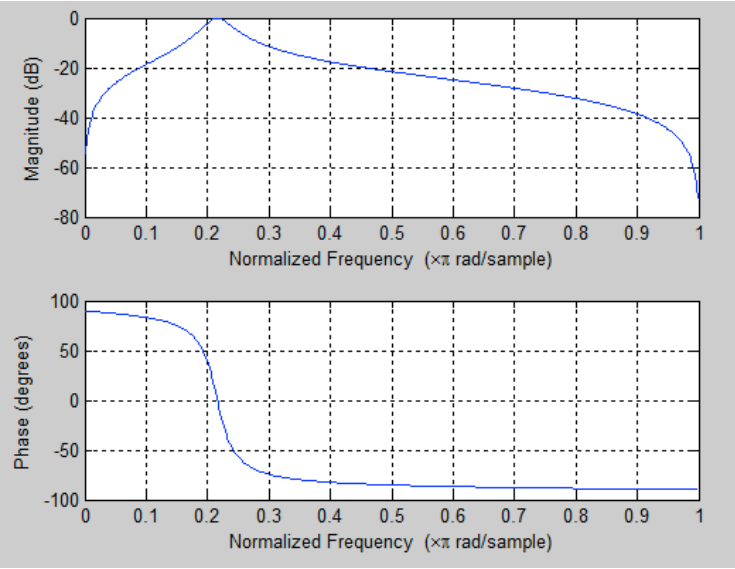


Figure 8. Bode and phase plot of a 5187Hz centered filter with a bandwidth of 1000Hz at a sampling frequency of 48000Hz.

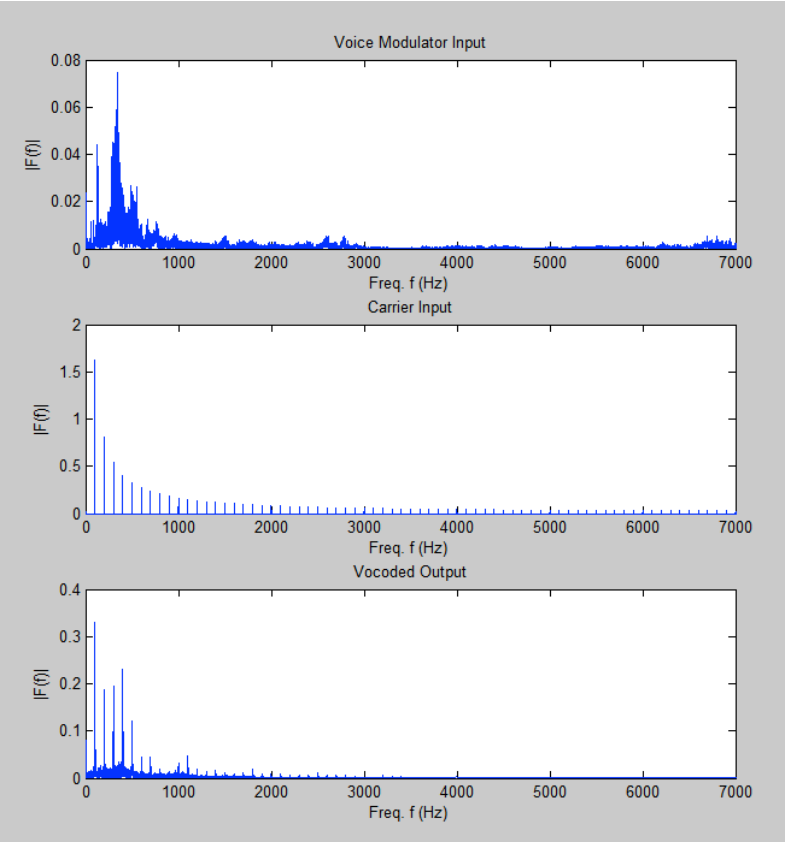


Figure 9. The Fourier spectra of an input human voice signal, a 100Hz sawtooth wave, and their vocoded output.

VHDL Implementation of MATLAB Code

While the translation from MATLAB simulations to VHDL code was trivial, getting the filters to work properly was an issue. Early on, we had suspected that there might be a problem during the conversion of the MATLAB code to VHDL and we were exactly right. The coefficients that describe digital filters are typically calculated out to be floating point numbers. Since the DE2 board cannot store decimal values, all the coefficients had to be multiplied by an integer value rounded to the nearest integer value. We tested the effect of this conversion in MATLAB and the result was that the output should still be exactly the same as its floating point relative.

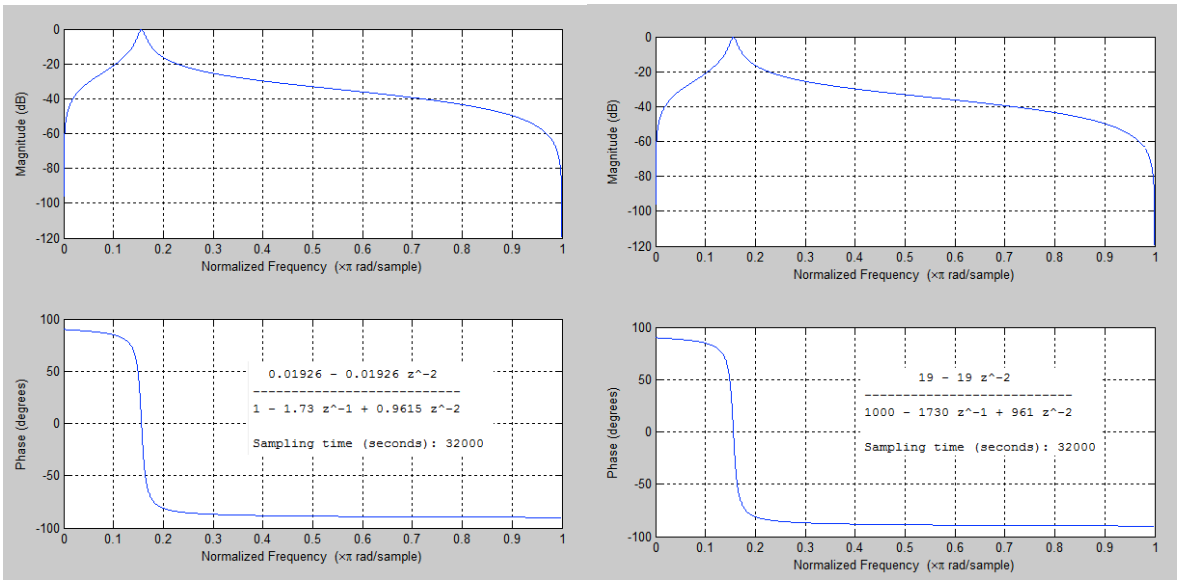


Figure 10. Side by side comparison of the floating point and fixed point representations of the same floating point output. The plots look exactly the same.

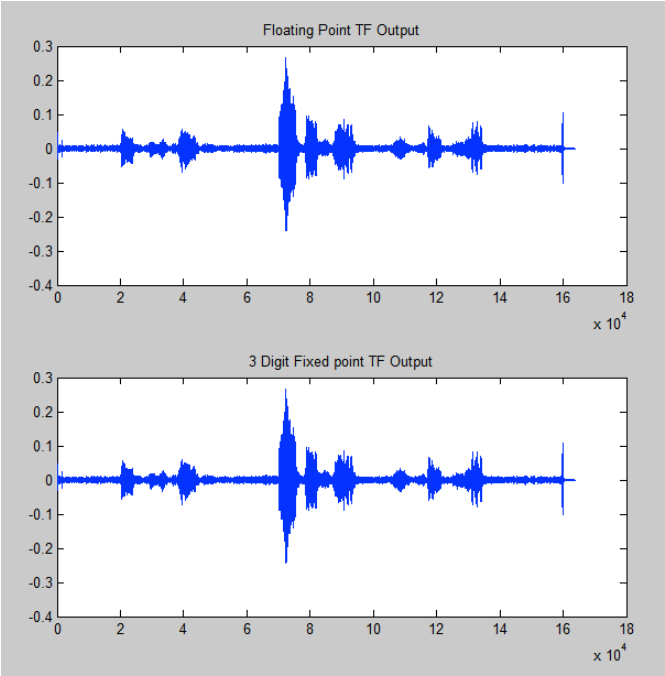
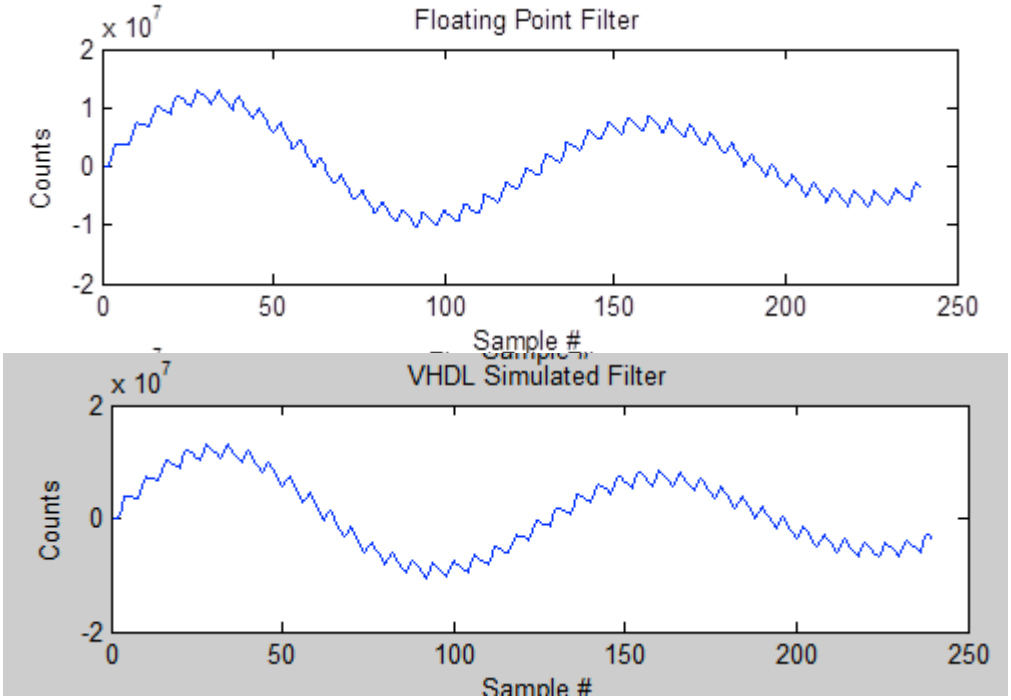


Figure 11. Side by side comparison of the same voice signal filtered by the floating point version and fixed point version of the same filter.

Although the simulations seemed to indicate that even with a fixed point filter implementation, we

would obtain a result that was faithful to the floating point filter, in the end, it seemed to indicate from our VHDL Functional Simulations that our filters failed to work properly due to diverging quantization errors. The floating point result would always be slightly higher than the fixed point output with the error accumulating more and more with each output sample. The cause of this undershooting in the actual implementation is that the division hardware is most likely just truncating any decimal values from integer division resulting in a floor function-esque result. The error accumulates because the transfer functions are implemented such that the next output sample is equal to a value that is comprised of the prior output, and the prior output before that. This means that any error accumulated at any given iteration of the function propagates to the next output. Although it may seem for the first few samples of a simulated VHDL filter that its output matches the MATLAB simulated filter outputs and is only off by a little, this error can and will grow to be so large that it overwhelms the magnitude of the true output signal.



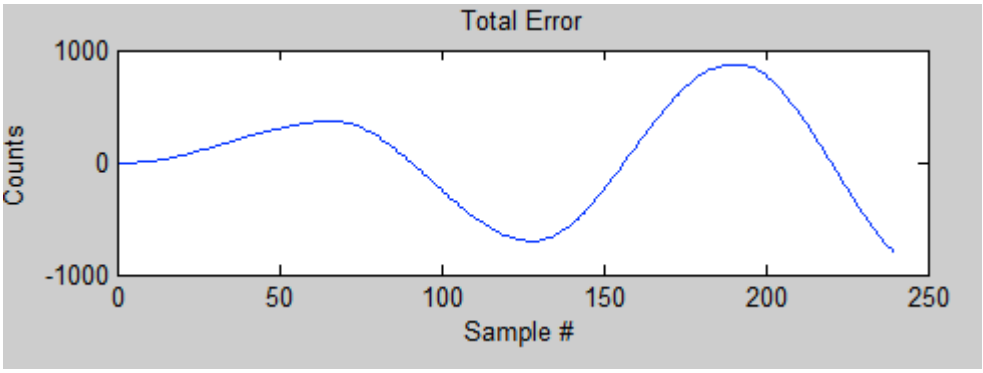


Figure 12. The Matlab simulated output, VHDL implementation simulated output, and error between the two outputs for the same given input signal. Although it may seem that the outputs are pretty much the same, as the number of samples fed into the system grows, so does the error. It is unbounded and divergent which is a really big problem.

We made several attempts to correct this error by adding a constant correction term to the output every two clock cycles but it was to no avail. We even suspected the root of the error and disparity to be caused by the one division operation in our implementation of the transfer function. We tried to correct this by normalizing the transfer function such that the divisor was a power of 2^N so that all that was required of the division was a shift to the right by N times. This helped to further decrease the error according to our VHDL simulations but unfortunately did not fix the issue of a diverging error.

3. Responsibilities and Lessons Learned

3.1 Responsibilities

Warren – Built and tested the MIDI conversion circuit; designed, simulated, and built filters in VHDL; worked on Vocoder implementation

Rob – Built the synthesizer logic, FM operators; partially responsible for interfacing the Nios with the synthesizer logic

Ashwin – Built the UART; Wrote the Nios software; partially responsible for interfacing the Nios with the synthesizer logic

3.2 Advice and Lessons Learned

We learned a lot about working on a large project with others. In other words, we learned more about the cooperation and coordination it takes for three different people to work on three different parts of one system and make it come together. One unexpected thing that we learned was not to underestimate the peripherals you work with – especially when it comes to trying to make things go together that were never intended to go together in the first place (the PS/2 interfacing with an RS-232-like protocol).

The best advice that we could give to future students would be to communicate with your teammates, have a clear idea of where you're going, and what you want. It's also extremely important to work on

interfacing with your peripherals early on. We found that the interfacing work presented us with much more frustration that just the straightforward VHDL and C coding on the whole (though those parts did manage to frustrate me too).

Appendix A. C Code Listings

1. *hello_world_small.c* – main program

```
#include <stdio.h>
#include <io.h>
#include <system.h>
#include "altera_avalon_pio_regs.h"
#include "note_frequencies.h"

#define NUMVOICES 5

int main() {
  unsigned char temp = 0;
  unsigned char byte = 0;
  unsigned char data[2] = {0, 0};

  unsigned char status = 0;
  unsigned char note = 0;
  unsigned char vel = 0;

  unsigned char onVoices[NUMVOICES] = {0};

  int numbytes = 0;
  int nextVoice = 0; // index of the next voice to add

  // clear any stray bytes that may be lingering in the register
  IOWR_ALTERA_AVALON_PIO_DATA(STATUS_PIO_BASE, 0xf1);

  // clear any sounds from the data registers
  IOWR_ALTERA_AVALON_PIO_DATA(NOTE_1_PIO_BASE, 0);
  IOWR_ALTERA_AVALON_PIO_DATA(NOTE_2_PIO_BASE, 0);
  IOWR_ALTERA_AVALON_PIO_DATA(NOTE_3_PIO_BASE, 0);
  IOWR_ALTERA_AVALON_PIO_DATA(NOTE_4_PIO_BASE, 0);
  IOWR_ALTERA_AVALON_PIO_DATA(NOTE_5_PIO_BASE, 0);

  //printf("Welcome to the Vocoder!\n");

  while (1) {
    IOWR_ALTERA_AVALON_PIO_DATA(STATUS_PIO_BASE, 0xf0); // disable
clearing
    temp = IORD_ALTERA_AVALON_PIO_DATA(STATUS_PIO_BASE); // read
status

    if (temp == 1) { // new byte available
        byte = IORD_ALTERA_AVALON_PIO_DATA(DATA_PIO_BASE); // read
data reg
        IOWR_ALTERA_AVALON_PIO_DATA(STATUS_PIO_BASE, 0xf1); // clear
status reg

        if (byte >> 7) { //status signal
```

```
            status = byte >> 4; // use only the top nibble (the
bottom nibble holds the channel)
            data[0] = 0;
            data[1] = 0;
            numbytes = 0;
        }
        else { // note-on or note-off byte
            data[numbytes] = byte;
            numbytes++;
        }
    }

    if (numbytes == 2) {
        note = data[0];
        vel = data[1];

        if (status == 0x9) { // note-on event
            if (vel != 0) { // true note-on
                if(((note - 21) >= 0) && ((note - 21) < 85)) { //
actual keyboard note
                    //int x = 0;
                    //while (x < NUMVOICES) {
                        //if ((onVoices[x] == 0)) {
                            //onVoices[x] = note;
                            onVoices[nextVoice] = note;
                            // turn note on
                            if( nextVoice == 0 )
                                playNote(note-21, NOTE_1_PIO_BASE);
                            else if( nextVoice == 1 )
                                playNote(note-21, NOTE_2_PIO_BASE);
                            else if( nextVoice == 2 )
                                playNote(note-21, NOTE_3_PIO_BASE);
                            else if( nextVoice == 3 )
                                playNote(note-21, NOTE_4_PIO_BASE);
                            else if( nextVoice == 4 )
                                playNote(note-21, NOTE_5_PIO_BASE);

                            if (nextVoice < 4)
                                nextVoice++;
                            else
                                nextVoice = 0;
                            //printf("Playing MIDI note: %d,
Frequency: %d\n", note, freq[note - 21]);
                           // break;
                        //}
                        //x++;
                    //}
                }
            }
            else { // actually a note-off
                if(((note - 21) >= 0) && ((note - 21) < 85)) { //
```

```
actual keyboard note
                    int x = 0;
                    while (x < NUMVOICES) {

                        if (onVoices[x] == note) {
                            onVoices[x] = 0;
                            // turn note off
                            if (x == 0)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_1_PIO_BASE, 0);
                            else if (x == 1)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_2_PIO_BASE, 0);
                            else if (x == 2)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_3_PIO_BASE, 0);
                            else if (x == 3)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_4_PIO_BASE, 0);
                            else if (x == 4)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_5_PIO_BASE, 0);
                            //printf("Stopping MIDI note: %d,
Frequency: %d\n", note, freq[note - 21]);
                            break;
                        }
                        x++;
                    }
                }
            }
        }
        else if (status == 0x8) { // explicit note-off event
            if(((note - 21) >= 0) && ((note - 21) < 85)) { // actual
keyboard note
                int x = 0;
                while (x < NUMVOICES) {
                    if (onVoices[x] == note) {
                        onVoices[x] = 0;
                        // turn note off
                        if (x == 0)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_1_PIO_BASE, 0);
                        else if (x == 1)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_2_PIO_BASE, 0);
                        else if (x == 2)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_3_PIO_BASE, 0);
                        else if (x == 3)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_4_PIO_BASE, 0);
```

```
                        else if (x == 4)

IOWR_ALTERA_AVALON_PIO_DATA(NOTE_5_PIO_BASE, 0);
                        //printf("Stopping MIDI note: %d, Frequency:
%d\n", note, freq[note - 21]);
                        break;
                    }
                    x++;
                }
            }
        }
        else {
            printf("Here\n");
        }
        numbytes = 0;
    }
  }
  return 0;
}
```

2. *note_frequencies.h* – note playing function

```
#ifndef NOTE_FREQUENCIES_H_
#define NOTE_FREQUENCIES_H_
#include "altera_avalon_pio_regs.h"

void playNote(int note, long base) {
switch (note) {
case 0: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x001b); break;
case 1: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x001d); break;
case 2: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x001e); break;
case 3: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0020); break;
case 4: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0022); break;
case 5: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0024); break;
case 6: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0026); break;
case 7: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0029); break;
case 8: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x002b); break;
case 9: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x002e); break;
case 10: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0030); break;
case 11: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0033); break;
case 12: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0037); break;
case 13: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x003a); break;
case 14: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x003d); break;
case 15: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0041); break;
case 16: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0045); break;
case 17: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0049); break;
case 18: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x004d); break;
case 19: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0052); break;
case 20: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0057); break;
case 21: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x005c); break;
```

```
case 22: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0061); break;
case 23: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0067); break;
case 24: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x006e); break;
case 25: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0074); break;
case 26: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x007b); break;
case 27: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0082); break;
case 28: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x008a); break;
case 29: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0092); break;
case 30: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x009b); break;
case 31: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00a4); break;
case 32: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00ae); break;
case 33: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00b8); break;
case 34: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00c3); break;
case 35: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00cf); break;
case 36: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00dc); break;
case 37: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00e9); break;
case 38: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x00f6); break;
case 39: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0105); break;
case 40: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0115); break;
case 41: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0125); break;
case 42: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0137); break;
case 43: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0149); break;
case 44: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x015d); break;
case 45: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0171); break;
case 46: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0187); break;
case 47: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x019f); break;
case 48: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x01b8); break;
case 49: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x01d2); break;
case 50: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x01ed); break;
case 51: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x020b); break;
case 52: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x022a); break;
case 53: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x024b); break;
case 54: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x026e); break;
case 55: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0293); break;
case 56: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x02ba); break;
case 57: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x02e3); break;
case 58: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x030f); break;
case 59: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x033e); break;
case 60: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0370); break;
case 61: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x03a4); break;
case 62: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x03db); break;
case 63: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0417); break;
case 64: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0455); break;
case 65: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0527); break;
case 66: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0575); break;
case 67: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x05c8); break;
case 68: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0620); break;
case 69: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x067d); break;
case 70: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x06e0); break;
case 71: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0749); break;
case 72: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x082d); break;
```

```
case 73: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x08aa); break;
case 74: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x092d); break;
case 75: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x09b9); break;
case 76: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0a4d); break;
case 77: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0ae9); break;
case 78: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0b90); break;
case 79: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0c40); break;
case 80: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0cfa); break;
case 81: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0dc0); break;
case 82: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0e91); break;
case 83: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x0f6f); break;
default: IOWR_ALTERA_AVALON_PIO_DATA(base, 0x105a); break;
}
}
#endif /*NOTE_FREQUENCIES_H_*/
```

Appendix B: VHDL Code Listings

Part I: midi_uart - Top Level Module

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

entity midi_uart is
     port (
          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);        -- (active
low)
          LEDG: out std_logic_vector(7 downto 0);            -- Green
LEDs (active high)
          LEDR : out std_logic_vector(17 downto 0);          -- Red LEDs
(active high)

          -- RS-232 interface

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

          -- 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 : in std_logic;       -- SD Card Data      SD pin 7
"DAT 0/DataOut"
          SD_DAT3 : out std_logic;     -- SD Card Data 3    SD pin 1
"DAT 3/nCS"
          SD_CMD : out std_logic;      -- SD Card Command   SD pin 2
"CMD/DataIn"
          SD_CLK : out std_logic;      -- SD Card Clock     SD pin 5
"CLK"

          -- 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 unsigned(15 downto 0);    -- DATA bus 16
Bits
          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

          SRAM_DQ : inout std_logic_vector(15 downto 0);
          SRAM_ADDR : out std_logic_vector(17 downto 0);
          SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N : out
std_logic
```

```
          );
end midi_uart;

architecture arch of midi_uart is
     -- signals for the MIDI decoder
     signal databyte, statusbyte : std_logic_vector (7 downto 0);
     signal writebyte : std_logic_vector (7 downto 0);

     signal note_data1 : std_logic_vector(15 downto 0);
     signal note_data2 : std_logic_vector(15 downto 0);
     signal note_data3 : std_logic_vector(15 downto 0);
     signal note_data4 : std_logic_vector(15 downto 0);
     signal note_data5 : std_logic_vector(15 downto 0);
     signal note_enable1, note_enable2, note_enable3, note_enable4,
note_enable5 : std_logic;

     --Synthesizer declaration
     component synthesizer is
     port(
          clk, reset_n : in std_logic;

          --FM synthesis inputs (from NIOS processor via Avalon bus)
          fm_dat_1, fm_dat_2, fm_dat_3, fm_dat_4, fm_dat_5 : in
std_logic_vector(15 downto 0);
          fm_en_1,  fm_en_2,  fm_en_3,  fm_en_4,  fm_en_5  : in
std_logic;

          --Select output (synthesizer or vocoder)
          select_out : std_logic;

          -- Switch data
          SW  : in std_logic_vector(17 downto 0);

          -- I2C bus (from pins)
          I2C_SDAT : inout std_logic; -- I2C Data
          I2C_SCLK : out std_logic;   -- I2C Clock

          --Audio CODEC (from pins)
          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
     );
```

```
    end component;

begin
          nios: entity work.nios_system port map (
          clk => CLOCK_50,
          reset_n => KEY(0),

          in_port_to_the_data_pio => databyte,

          out_port_from_the_status_pio => writebyte,
          in_port_to_the_status_pio => statusbyte,

          --note info

        out_port_from_the_note_1_pio => note_data1,
          out_port_from_the_note_2_pio => note_data2,
          out_port_from_the_note_3_pio => note_data3,
          out_port_from_the_note_4_pio => note_data4,
          out_port_from_the_note_5_pio => note_data5,

          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
          );

          -- Set note enables
          note_enable1 <= '0' when note_data1 = x"0000" else '1';

          note_enable2 <= '0' when note_data2 = x"0000" else '1';
          note_enable3 <= '0' when note_data3 = x"0000" else '1';
          note_enable4 <= '0' when note_data4 = x"0000" else '1';
          note_enable5 <= '0' when note_data5 = x"0000" else '1';


          -- Instantiate synthesizer
          syn : synthesizer port map (
               clk => CLOCK_50,
               reset_n => KEY(0),

               select_out => SW(0),

               -- pitch data
               --fm_dat_1 => x"01B8",
               fm_dat_1 => note_data1,
               fm_dat_2 => note_data2,
               fm_dat_3 => note_data3,
               fm_dat_4 => note_data4,
```

```
      fm_dat_5 => note_data5,

      -- voice enable
      fm_en_1  => note_enable1,
      fm_en_2  => note_enable2,
      fm_en_3  => note_enable3,
      fm_en_4  => note_enable4,
      fm_en_5  => note_enable5,

      -- Switch data
      SW => SW,

      -- I2C bus (from pins)
      I2C_SDAT => I2C_SDAT,
      I2C_SCLK => I2C_SCLK,

      --Audio CODEC (from pins)
      AUD_ADCLRCK => AUD_ADCLRCK,
      AUD_ADCDAT  => AUD_ADCDAT,
      AUD_DACLRCK => AUD_DACLRCK,
      AUD_DACDAT  => AUD_DACDAT,
      AUD_BCLK  => AUD_BCLK,
      AUD_XCK     => AUD_XCK
);

the_uart : entity work.Midi_interface port map (
clk => CLOCK_50,
rst => not KEY(0),
rx => PS2_DAT,
clr_flag => writebyte(0),
data_out => databyte,
status_out => statusbyte
);

HEX7     <= "0001001"; -- Leftmost
HEX6     <= "0000110";
HEX5     <= "1000111";
HEX4     <= "1000111";
HEX3     <= "1000000";
HEX2     <= (others => '1');
HEX1     <= (others => '1');
HEX0     <= (others => '1');          -- Rightmost

LEDG  <= databyte;

LEDR <= SW;

LCD_ON   <= '1';
LCD_BLON <= '1';
LCD_RW <= '1';
LCD_EN <= '0';
```

```
LCD_RS <= '0';

VGA_CLK <= '0';
VGA_HS <= '0';
VGA_VS <= '0';
VGA_BLANK <= '0';
VGA_SYNC <= '0';
VGA_R <= (others => '0');
VGA_G <= (others => '0');
VGA_B <= (others => '0');

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

--I2C_SCLK <= '0';

ENET_CMD <= '0';
ENET_CS_N <= '1';
ENET_WR_N <= '1';
ENET_RD_N <= '1';
```

```
        ENET_RST_N <= '1';
        ENET_CLK <= '0';

        --AUD_DACDAT <= '0';
        --AUD_XCK <= '0';

        TD_RESET <= '0';

        -- Set all bidirectional ports to tri-state
        DRAM_DQ     <= (others => 'Z');
        FL_DQ       <= (others => 'Z');
        OTG_DATA    <= (others => 'Z');
        LCD_DATA    <= (others => '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 arch;
```

Part II: UART VHDL Files

1. tick_counter.vhd – slowed down clock entity for baud rate counting

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

entity tick_counter is
     port (
          clk, rst: in std_logic;
          tick: out std_logic
     );
end tick_counter;

architecture arch of tick_counter is
     signal current_val : unsigned (7 downto 0);
     signal next_val    : unsigned (7 downto 0);

     begin
     process (clk)
          begin
          if rising_edge(clk) then
               if(rst = '1') then
                    current_val <= (others => '0');
               else
                    current_val <= next_val;
               end if;
```

```
          end if;
     end process;

     next_val <= (others => '0') when current_val = 99 else
current_val + 1;
     tick <= '1' when current_val = 99 else '0';
end arch;
```

2. uart_rec.vhd – state machine that speaks the MIDI protocol and reads in a byte

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

entity uart_rec is
     port (
          clk, rst : in std_logic;
          rx : in std_logic;
          tick : in std_logic;
          data_ready : out std_logic;
          data_out : out std_logic_vector (7 downto 0)
     );
end uart_rec;

architecture arch of uart_rec is
     type states is (idle, start, data, stop);
     signal this_state, next_state : states; -- state of the FSM
     signal this_count, next_count : unsigned (3 downto 0); -- the
number of baud samples taken (0 - 15)
     signal this_bits, next_bits : unsigned (3 downto 0); -- the
number of bits received (0 - 7)
     signal this_data, next_data : std_logic_vector (7 downto 0); --
the actual incoming data

     begin
     process (clk) -- sensitive to the system clock
          begin
          if rising_edge(clk) then
               if(rst = '1') then -- reset signals
                    this_state <= idle;
                    this_count <= (others => '0');
                    this_bits <= (others => '0');
                    this_data <= (others => '0');
               else -- assign the current signals to their next value
                    this_state <= next_state;
                    this_count <= next_count;
                    this_bits <= next_bits;
                    this_data <= next_data;
               end if;
          end if;
     end process;
```

```
     process (this_state, this_count, this_bits, this_data, tick, rx)
-- whenever a signal or input (tick, rx) changes
     begin
          -- defensively assign the next signal values as the current
signal values
          next_data <= this_data;
          next_state <= this_state;
          next_count <= this_count;
          next_bits <= this_bits;
          data_ready <= '0'; -- data isn't ready
          case this_state is
               when idle =>
                    if (rx = '0') then -- sign of a start signal bit
                         next_state <= start; -- move to start state
                         next_count <= (others => '0'); -- reset
counter
                    end if;
               when start =>
                    if (tick = '1') then -- align ourselves to the
baudrate counter
                         if (this_count = 7) then -- if we are in the
middle of the start bit (7 samples in)
                              next_state <= data; -- move to data
state
                              next_data <= (others => '0'); -- reset
data byte
                              next_count <= (others => '0'); -- reset
counter
                              next_bits <= (others => '0'); -- reset
the number of bits read
                         else -- increment counter
                              next_count <= this_count + 1;
                         end if;
                    end if;
               when data =>
                    if (tick = '1') then -- align ourselves to the
baudrate counter
                         if (this_count = 15) then -- if we count 15
samples of the input
                              next_count <= (others => '0'); -- reset
the counter (for the next step)
                              next_data <= rx & this_data(7 downto
1); -- shift the LSB out and add rx as the MSB of the data
                              if (this_bits = 7) then -- we have read
a full byte
                                   next_state <= stop; -- move to
stop stage
                              else -- we need to read another byte
                                   next_bits <= this_bits + 1; --
increment our bit count
```

```
                              end if;
                         else -- we haven't taken enough samples of
the signal
                              next_count <= this_count + 1; --
increment counter
                         end if;
                    end if;
               when stop =>
                    if (tick = '1') then -- align ourselves to the
baudrate counter
                         if (this_count = 15) then -- if we count 15
samples of the input
                              next_state <= idle; -- move back to the
idle state
                              data_ready <= '1'; -- assert the "done"
status signal high
                         else -- we haven't taken enough samples of
the signal
                              next_count <= this_count + 1; --
increment counter
                         end if;
                    end if;
          end case;
     end process;
     data_out <= this_data; -- update output
end arch;
```

3. byte_reg.vhd – the 1 byte register to hold the byte

```
library ieee;
use ieee.std_logic_1164.all;

entity byte_reg is
     port (
          clk, rst : in std_logic;
          clr_flag, set_flag : in std_logic;
          data_in : in std_logic_vector (7 downto 0);
          data_out : out std_logic_vector (7 downto 0);
          flag : out std_logic
     );
end byte_reg;

architecture arch of byte_reg is
     signal curr_buff, next_buff : std_logic_vector (7 downto 0); --
signals for current and next byte
     signal curr_flag, next_flag : std_logic; -- signals for current
and next flag

     begin
     process (clk)
          begin
```

```
          if rising_edge(clk) then
               if(rst = '1') then -- reset all signals
                    curr_buff <= (others => '0');
                    curr_flag <= '0';
               else -- set current signals to their next values
                    curr_buff <= next_buff;
                    curr_flag <= next_flag;
               end if;
          end if;
     end process;

     process (curr_buff, curr_flag, set_flag, clr_flag, data_in) --
for updating values
          begin
          -- defensively define the values for next_buff and
next_flag
          next_buff <= curr_buff;
          next_flag <= curr_flag;
          if (set_flag = '1') then -- a new data value is available
               next_buff <= data_in; -- store the byte
               next_flag <= '1'; -- set the flag
          elsif (clr_flag = '1') then -- the processor wants to clear
the flag
               next_flag <= '0'; -- clear the flag
          end if;
     end process;

     -- output logic
     data_out <= curr_buff;
     flag <= curr_flag;
end arch;
```

4. Midi_interface – the top level module of the UART

```
library ieee;
use ieee.std_logic_1164.all;

entity Midi_interface is
     port (
          clk, rst : in std_logic;
          rx : in std_logic;
          clr_flag : in std_logic;
          data_out : out std_logic_vector (7 downto 0);
          status_out : out std_logic_vector (7 downto 0)
     );
end Midi_interface;

architecture arch of Midi_interface is
     component tick_counter
     port (
          clk, rst: in std_logic;
```

```
          tick: out std_logic
     );
     end component;

     component uart_rec
     port (
          clk, rst : in std_logic;
          rx : in std_logic;
          tick : in std_logic;
          data_ready : out std_logic;
          data_out : out std_logic_vector (7 downto 0)
     );
     end component;

     component byte_reg
     port (
          clk, rst : in std_logic;
          clr_flag, set_flag : in std_logic;
          data_in : in std_logic_vector (7 downto 0);
          data_out : out std_logic_vector (7 downto 0);
          flag : out std_logic
     );
     end component;

     signal tick_sig, done_sig : std_logic;
     signal the_byte, dat, curr_out, next_out: std_logic_vector (7
downto 0);
     signal curr_stat, next_stat : std_logic_vector (7 downto 0);
     signal flag_sig : std_logic;

     begin

     ctr: tick_counter port map (clk, rst, tick_sig);
     uart: uart_rec port map (clk, rst, rx, tick_sig, done_sig,
the_byte);
     reg: byte_reg port map (clk, rst, clr_flag, done_sig, the_byte,
dat, flag_sig);

     process (flag_sig, dat, curr_out)
     begin
          next_out <= dat;
          next_stat <= "0000000" & flag_sig;
     end process;

     process (clk)
     begin
          if rst = '1' then
               curr_out <= (others => '0');
               curr_stat <= (others => '0');
          elsif rising_edge(clk) then
               curr_out <= next_out;
```

```
                curr_stat <= next_stat;
            end if;
        end process;

        data_out <= curr_out;
        status_out <= curr_stat;
end arch;
```

Part III: Synthesizer VHDL Files

1. fmOperatior.vhd – the FM operator module

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

entity fmOperator is
port (
    clk : in std_logic;
    reset_n : in std_logic;
     mute    : in std_logic;
     -- Oscillator select:
     --   0 = sine
     --  1 = saw
     --  2 = square
     --  3 = white noise (random)
     osc_select   : in std_logic_vector(1 downto 0);

     -- Oscillator period
     osc_freq   : in unsigned(15 downto 0);

     -- Modulator
     modulator  : in unsigned(23 downto 0);
     mod_factor : in unsigned(4 downto 0);

     -- Sample out
     fm_out   : out std_logic_vector(23 downto 0)
  );
end fmOperator;

architecture rtl of fmOperator is

    signal sin_out     : unsigned(23 downto 0);
     signal saw_out     : unsigned(23 downto 0);
     signal random_out  : unsigned(23 downto 0);
     signal square_out  : unsigned(23 downto 0);

     signal pre_out     : unsigned(23 downto 0);

    signal sin_counter : unsigned(8 downto 0);
     signal square_sel  : unsigned(0 downto 0);
```

```
     signal freq_counter: unsigned(19 downto 0);
     signal conversion  : unsigned(19 downto 0) := x"17D78";
     --signal conversion  : unsigned(19 downto 0) := x"0BEBC";
     signal temp           : unsigned(19 downto 0);
begin

     -- Generate wave form
     with osc_select select pre_out <=
          sin_out when "00",
          saw_out when "01",
          square_out when "10",
          random_out when "11",
          sin_out when others;

     fm_out <= std_logic_vector(pre_out) when mute = '0' else
x"000000";

     saw_out <= "000" & sin_counter & x"000";

     square_out <= x"011111" when square_sel = "1" else x"000000";

     temp <= conversion / (osc_freq+(shift_right(modulator,
TO_INTEGER(mod_factor))));

     process(clk)
       begin
          if rising_edge(clk) then
               if reset_n = '0' then
                    sin_counter <= (others => '0');
                    freq_counter <= (others => '0');
               else
                    if freq_counter = temp then
                         freq_counter <= (others => '0');
                         sin_counter <= sin_counter + 1;

                         square_sel <= not square_sel;
                    else
                         freq_counter <= freq_counter + 1;
                    end if;

               end if;
          end if;
     end process;

--256 sin samples, one complete oscillation
with sin_counter select sin_out <=
X"000000" when "000000000",
X"003243" when "000000001",
X"006485" when "000000010",
X"0096c3" when "000000011",
```

```
X"00c8fb" when "000000100",
X"00fb2b" when "000000101",
X"012d51" when "000000110",
X"015f6c" when "000000111",
X"01917a" when "000001000",
X"01c378" when "000001001",
X"01f564" when "000001010",
X"02273d" when "000001011",
X"025901" when "000001100",
X"028aae" when "000001101",
X"02bc42" when "000001110",
X"02edbb" when "000001111",
X"031f16" when "000010000",
X"035053" when "000010001",
X"038170" when "000010010",
X"03b269" when "000010011",
X"03e33e" when "000010100",
X"0413ed" when "000010101",
X"044474" when "000010110",
X"0474d0" when "000010111",
X"04a501" when "000011000",
X"04d503" when "000011001",
X"0504d6" when "000011010",
X"053478" when "000011011",
X"0563e6" when "000011100",
X"05931f" when "000011101",
X"05c220" when "000011110",
X"05f0e9" when "000011111",
X"061f78" when "000100000",
X"064dca" when "000100001",
X"067bdd" when "000100010",
X"06a9b1" when "000100011",
X"06d743" when "000100100",
X"070492" when "000100101",
X"07319b" when "000100110",
X"075e5d" when "000100111",
X"078ad6" when "000101000",
X"07b705" when "000101001",
X"07e2e8" when "000101010",
X"080e7d" when "000101011",
X"0839c3" when "000101100",
X"0864b7" when "000101101",
X"088f59" when "000101110",
X"08b9a6" when "000101111",
X"08e39d" when "000110000",
X"090d3c" when "000110001",
X"093682" when "000110010",
X"095f6c" when "000110011",
X"0987fb" when "000110100",
X"09b02b" when "000110101",
X"09d7fc" when "000110110",
```

```
X"09ff6c" when "000110111",
X"0a2678" when "000111000",
X"0a4d21" when "000111001",
X"0a7364" when "000111010",
X"0a9940" when "000111011",
X"0abeb3" when "000111100",
X"0ae3bd" when "000111101",
X"0b085a" when "000111110",
X"0b2c8b" when "000111111",
X"0b504e" when "001000000",
X"0b73a1" when "001000001",
X"0b9683" when "001000010",
X"0bb8f2" when "001000011",
X"0bdaee" when "001000100",
X"0bfc75" when "001000101",
X"0c1d86" when "001000110",
X"0c3e1f" when "001000111",
X"0c5e3f" when "001001000",
X"0c7de5" when "001001001",
X"0c9d10" when "001001010",
X"0cbbbe" when "001001011",
X"0cd9ef" when "001001100",
X"0cf7a1" when "001001101",
X"0d14d2" when "001001110",
X"0d3183" when "001001111",
X"0d4db2" when "001010000",
X"0d695d" when "001010001",
X"0d8484" when "001010010",
X"0d9f25" when "001010011",
X"0db940" when "001010100",
X"0dd2d4" when "001010101",
X"0debdf" when "001010110",
X"0e0461" when "001010111",
X"0e1c58" when "001011000",
X"0e33c4" when "001011001",
X"0e4aa4" when "001011010",
X"0e60f7" when "001011011",
X"0e76bc" when "001011100",
X"0e8bf2" when "001011101",
X"0ea099" when "001011110",
X"0eb4af" when "001011111",
X"0ec834" when "001100000",
X"0edb28" when "001100001",
X"0eed88" when "001100010",
X"0eff56" when "001100011",
X"0f108f" when "001100100",
X"0f2134" when "001100101",
X"0f3143" when "001100110",
X"0f40bc" when "001100111",
X"0f4f9f" when "001101000",
X"0f5deb" when "001101001",
```

```
X"0f6b9f" when "001101010",
X"0f78bb" when "001101011",
X"0f853e" when "001101100",
X"0f9128" when "001101101",
X"0f9c78" when "001101110",
X"0fa72f" when "001101111",
X"0fb14a" when "001110000",
X"0fbacb" when "001110001",
X"0fc3b1" when "001110010",
X"0fcbfb" when "001110011",
X"0fd3a9" when "001110100",
X"0fdabb" when "001110101",
X"0fe131" when "001110110",
X"0fe70a" when "001110111",
X"0fec45" when "001111000",
X"0ff0e4" when "001111001",
X"0ff4e5" when "001111010",
X"0ff849" when "001111011",
X"0ffb0f" when "001111100",
X"0ffd38" when "001111101",
X"0ffec3" when "001111110",
X"0fffb0" when "001111111",
X"0fffff" when "010000000",
X"0fffb0" when "010000001",
X"0ffec3" when "010000010",
X"0ffd38" when "010000011",
X"0ffb0f" when "010000100",
X"0ff849" when "010000101",
X"0ff4e5" when "010000110",
X"0ff0e4" when "010000111",
X"0fec45" when "010001000",
X"0fe70a" when "010001001",
X"0fe131" when "010001010",
X"0fdabb" when "010001011",
X"0fd3a9" when "010001100",
X"0fcbfb" when "010001101",
X"0fc3b1" when "010001110",
X"0fbacb" when "010001111",
X"0fb14a" when "010010000",
X"0fa72f" when "010010001",
X"0f9c78" when "010010010",
X"0f9128" when "010010011",
X"0f853e" when "010010100",
X"0f78bb" when "010010101",
X"0f6b9f" when "010010110",
X"0f5deb" when "010010111",
X"0f4f9f" when "010011000",
X"0f40bc" when "010011001",
X"0f3143" when "010011010",
X"0f2134" when "010011011",
X"0f108f" when "010011100",
```

```
X"0eff56" when "010011101",
X"0eed88" when "010011110",
X"0edb28" when "010011111",
X"0ec834" when "010100000",
X"0eb4af" when "010100001",
X"0ea099" when "010100010",
X"0e8bf2" when "010100011",
X"0e76bc" when "010100100",
X"0e60f7" when "010100101",
X"0e4aa4" when "010100110",
X"0e33c4" when "010100111",
X"0e1c58" when "010101000",
X"0e0461" when "010101001",
X"0debdf" when "010101010",
X"0dd2d4" when "010101011",
X"0db940" when "010101100",
X"0d9f25" when "010101101",
X"0d8484" when "010101110",
X"0d695d" when "010101111",
X"0d4db2" when "010110000",
X"0d3183" when "010110001",
X"0d14d2" when "010110010",
X"0cf7a1" when "010110011",
X"0cd9ef" when "010110100",
X"0cbbbe" when "010110101",
X"0c9d10" when "010110110",
X"0c7de5" when "010110111",
X"0c5e3f" when "010111000",
X"0c3e1f" when "010111001",
X"0c1d86" when "010111010",
X"0bfc75" when "010111011",
X"0bdaee" when "010111100",
X"0bb8f2" when "010111101",
X"0b9683" when "010111110",
X"0b73a1" when "010111111",
X"0b504e" when "011000000",
X"0b2c8b" when "011000001",
X"0b085a" when "011000010",
X"0ae3bd" when "011000011",
X"0abeb3" when "011000100",
X"0a9940" when "011000101",
X"0a7364" when "011000110",
X"0a4d21" when "011000111",
X"0a2678" when "011001000",
X"09ff6c" when "011001001",
X"09d7fc" when "011001010",
X"09b02b" when "011001011",
X"0987fb" when "011001100",
X"095f6c" when "011001101",
X"093682" when "011001110",
X"090d3c" when "011001111",
```

```
X"08e39d" when "011010000",
X"08b9a6" when "011010001",
X"088f59" when "011010010",
X"0864b7" when "011010011",
X"0839c3" when "011010100",
X"080e7d" when "011010101",
X"07e2e8" when "011010110",
X"07b705" when "011010111",
X"078ad6" when "011011000",
X"075e5d" when "011011001",
X"07319b" when "011011010",
X"070492" when "011011011",
X"06d743" when "011011100",
X"06a9b1" when "011011101",
X"067bdd" when "011011110",
X"064dca" when "011011111",
X"061f78" when "011100000",
X"05f0e9" when "011100001",
X"05c220" when "011100010",
X"05931f" when "011100011",
X"0563e6" when "011100100",
X"053478" when "011100101",
X"0504d6" when "011100110",
X"04d503" when "011100111",
X"04a501" when "011101000",
X"0474d0" when "011101001",
X"044474" when "011101010",
X"0413ed" when "011101011",
X"03e33e" when "011101100",
X"03b269" when "011101101",
X"038170" when "011101110",
X"035053" when "011101111",
X"031f16" when "011110000",
X"02edbb" when "011110001",
X"02bc42" when "011110010",
X"028aae" when "011110011",
X"025901" when "011110100",
X"02273d" when "011110101",
X"01f564" when "011110110",
X"01c378" when "011110111",
X"01917a" when "011111000",
X"015f6c" when "011111001",
X"012d51" when "011111010",
X"00fb2b" when "011111011",
X"00c8fb" when "011111100",
X"0096c3" when "011111101",
X"006485" when "011111110",
X"003243" when "011111111",
X"000000" when "100000000",
X"ffcdbd" when "100000001",
X"ff9b7b" when "100000010",
```

```
X"ff693d" when "100000011",
X"ff3705" when "100000100",
X"ff04d5" when "100000101",
X"fed2af" when "100000110",
X"fea094" when "100000111",
X"fe6e86" when "100001000",
X"fe3c88" when "100001001",
X"fe0a9c" when "100001010",
X"fdd8c3" when "100001011",
X"fda6ff" when "100001100",
X"fd7552" when "100001101",
X"fd43be" when "100001110",
X"fd1245" when "100001111",
X"fce0ea" when "100010000",
X"fcafad" when "100010001",
X"fc7e90" when "100010010",
X"fc4d97" when "100010011",
X"fc1cc2" when "100010100",
X"fbec13" when "100010101",
X"fbbb8c" when "100010110",
X"fb8b30" when "100010111",
X"fb5aff" when "100011000",
X"fb2afd" when "100011001",
X"fafb2a" when "100011010",
X"facb88" when "100011011",
X"fa9c1a" when "100011100",
X"fa6ce1" when "100011101",
X"fa3de0" when "100011110",
X"fa0f17" when "100011111",
X"f9e088" when "100100000",
X"f9b236" when "100100001",
X"f98423" when "100100010",
X"f9564f" when "100100011",
X"f928bd" when "100100100",
X"f8fb6e" when "100100101",
X"f8ce65" when "100100110",
X"f8a1a3" when "100100111",
X"f8752a" when "100101000",
X"f848fb" when "100101001",
X"f81d18" when "100101010",
X"f7f183" when "100101011",
X"f7c63d" when "100101100",
X"f79b49" when "100101101",
X"f770a7" when "100101110",
X"f7465a" when "100101111",
X"f71c63" when "100110000",
X"f6f2c4" when "100110001",
X"f6c97e" when "100110010",
X"f6a094" when "100110011",
X"f67805" when "100110100",
X"f64fd5" when "100110101",
```

```
X"f62804" when "100110110",
X"f60094" when "100110111",
X"f5d988" when "100111000",
X"f5b2df" when "100111001",
X"f58c9c" when "100111010",
X"f566c0" when "100111011",
X"f5414d" when "100111100",
X"f51c43" when "100111101",
X"f4f7a6" when "100111110",
X"f4d375" when "100111111",
X"f4afb2" when "101000000",
X"f48c5f" when "101000001",
X"f4697d" when "101000010",
X"f4470e" when "101000011",
X"f42512" when "101000100",
X"f4038b" when "101000101",
X"f3e27a" when "101000110",
X"f3c1e1" when "101000111",
X"f3a1c1" when "101001000",
X"f3821b" when "101001001",
X"f362f0" when "101001010",
X"f34442" when "101001011",
X"f32611" when "101001100",
X"f3085f" when "101001101",
X"f2eb2e" when "101001110",
X"f2ce7d" when "101001111",
X"f2b24e" when "101010000",
X"f296a3" when "101010001",
X"f27b7c" when "101010010",
X"f260db" when "101010011",
X"f246c0" when "101010100",
X"f22d2c" when "101010101",
X"f21421" when "101010110",
X"f1fb9f" when "101010111",
X"f1e3a8" when "101011000",
X"f1cc3c" when "101011001",
X"f1b55c" when "101011010",
X"f19f09" when "101011011",
X"f18944" when "101011100",
X"f1740e" when "101011101",
X"f15f67" when "101011110",
X"f14b51" when "101011111",
X"f137cc" when "101100000",
X"f124d8" when "101100001",
X"f11278" when "101100010",
X"f100aa" when "101100011",
X"f0ef71" when "101100100",
X"f0decc" when "101100101",
X"f0cebd" when "101100110",
X"f0bf44" when "101100111",
X"f0b061" when "101101000",
```

```
X"f0a215" when "101101001",
X"f09461" when "101101010",
X"f08745" when "101101011",
X"f07ac2" when "101101100",
X"f06ed8" when "101101101",
X"f06388" when "101101110",
X"f058d1" when "101101111",
X"f04eb6" when "101110000",
X"f04535" when "101110001",
X"f03c4f" when "101110010",
X"f03405" when "101110011",
X"f02c57" when "101110100",
X"f02545" when "101110101",
X"f01ecf" when "101110110",
X"f018f6" when "101110111",
X"f013bb" when "101111000",
X"f00f1c" when "101111001",
X"f00b1b" when "101111010",
X"f007b7" when "101111011",
X"f004f1" when "101111100",
X"f002c8" when "101111101",
X"f0013d" when "101111110",
X"f00050" when "101111111",
X"f00001" when "110000000",
X"f00050" when "110000001",
X"f0013d" when "110000010",
X"f002c8" when "110000011",
X"f004f1" when "110000100",
X"f007b7" when "110000101",
X"f00b1b" when "110000110",
X"f00f1c" when "110000111",
X"f013bb" when "110001000",
X"f018f6" when "110001001",
X"f01ecf" when "110001010",
X"f02545" when "110001011",
X"f02c57" when "110001100",
X"f03405" when "110001101",
X"f03c4f" when "110001110",
X"f04535" when "110001111",
X"f04eb6" when "110010000",
X"f058d1" when "110010001",
X"f06388" when "110010010",
X"f06ed8" when "110010011",
X"f07ac2" when "110010100",
X"f08745" when "110010101",
X"f09461" when "110010110",
X"f0a215" when "110010111",
X"f0b061" when "110011000",
X"f0bf44" when "110011001",
X"f0cebd" when "110011010",
X"f0decc" when "110011011",
```

```
X"f0ef71" when "110011100",
X"f100aa" when "110011101",
X"f11278" when "110011110",
X"f124d8" when "110011111",
X"f137cc" when "110100000",
X"f14b51" when "110100001",
X"f15f67" when "110100010",
X"f1740e" when "110100011",
X"f18944" when "110100100",
X"f19f09" when "110100101",
X"f1b55c" when "110100110",
X"f1cc3c" when "110100111",
X"f1e3a8" when "110101000",
X"f1fb9f" when "110101001",
X"f21421" when "110101010",
X"f22d2c" when "110101011",
X"f246c0" when "110101100",
X"f260db" when "110101101",
X"f27b7c" when "110101110",
X"f296a3" when "110101111",
X"f2b24e" when "110110000",
X"f2ce7d" when "110110001",
X"f2eb2e" when "110110010",
X"f3085f" when "110110011",
X"f32611" when "110110100",
X"f34442" when "110110101",
X"f362f0" when "110110110",
X"f3821b" when "110110111",
X"f3a1c1" when "110111000",
X"f3c1e1" when "110111001",
X"f3e27a" when "110111010",
X"f4038b" when "110111011",
X"f42512" when "110111100",
X"f4470e" when "110111101",
X"f4697d" when "110111110",
X"f48c5f" when "110111111",
X"f4afb2" when "111000000",
X"f4d375" when "111000001",
X"f4f7a6" when "111000010",
X"f51c43" when "111000011",
X"f5414d" when "111000100",
X"f566c0" when "111000101",
X"f58c9c" when "111000110",
X"f5b2df" when "111000111",
X"f5d988" when "111001000",
X"f60094" when "111001001",
X"f62804" when "111001010",
X"f64fd5" when "111001011",
X"f67805" when "111001100",
X"f6a094" when "111001101",
X"f6c97e" when "111001110",
```

```
X"f6f2c4" when "111001111",
X"f71c63" when "111010000",
X"f7465a" when "111010001",
X"f770a7" when "111010010",
X"f79b49" when "111010011",
X"f7c63d" when "111010100",
X"f7f183" when "111010101",
X"f81d18" when "111010110",
X"f848fb" when "111010111",
X"f8752a" when "111011000",
X"f8a1a3" when "111011001",
X"f8ce65" when "111011010",
X"f8fb6e" when "111011011",
X"f928bd" when "111011100",
X"f9564f" when "111011101",
X"f98423" when "111011110",
X"f9b236" when "111011111",
X"f9e088" when "111100000",
X"fa0f17" when "111100001",
X"fa3de0" when "111100010",
X"fa6ce1" when "111100011",
X"fa9c1a" when "111100100",
X"facb88" when "111100101",
X"fafb2a" when "111100110",
X"fb2afd" when "111100111",
X"fb5aff" when "111101000",
X"fb8b30" when "111101001",
X"fbbb8c" when "111101010",
X"fbec13" when "111101011",
X"fc1cc2" when "111101100",
X"fc4d97" when "111101101",
X"fc7e90" when "111101110",
X"fcafad" when "111101111",
X"fce0ea" when "111110000",
X"fd1245" when "111110001",
X"fd43be" when "111110010",
X"fd7552" when "111110011",
X"fda6ff" when "111110100",
X"fdd8c3" when "111110101",
X"fe0a9c" when "111110110",
X"fe3c88" when "111110111",
X"fe6e86" when "111111000",
X"fea094" when "111111001",
X"fed2af" when "111111010",
X"ff04d5" when "111111011",
X"ff3705" when "111111100",
X"ff693d" when "111111101",
X"ff9b7b" when "111111110",
X"ffcdbd" when "111111111",
X"000000" when others;
```

```
with sin_counter(7 downto 0) select random_out <=
X"06b62d" when "00000000",
X"0904e0" when "00000001",
X"08c3ee" when "00000010",
X"03fcbc" when "00000011",
X"089923" when "00000100",
X"0bcfbd" when "00000101",
X"08e090" when "00000110",
X"01856f" when "00000111",
X"0dbae2" when "00001000",
X"031556" when "00001001",
X"0aa0e6" when "00001010",
X"058dce" when "00001011",
X"0fa46a" when "00001100",
X"06b14f" when "00001101",
X"09e375" when "00001110",
X"059a82" when "00001111",
X"0570c5" when "00010000",
X"004ec7" when "00010001",
X"08c0b0" when "00010010",
X"043d34" when "00010011",
X"0b6731" when "00010100",
X"0631cd" when "00010101",
X"0976b8" when "00010110",
X"0f35bc" when "00010111",
X"092b5d" when "00011000",
X"0e6163" when "00011001",
X"0e177f" when "00011010",
X"0240b1" when "00011011",
X"095157" when "00011100",
X"08c8cb" when "00011101",
X"0c00ed" when "00011110",
X"0bf40f" when "00011111",
X"0536da" when "00100000",
X"0f1aad" when "00100001",
X"0e7ab4" when "00100010",
X"017e85" when "00100011",
X"069781" when "00100100",
X"0bf4b4" when "00100101",
X"07a95c" when "00100110",
X"081214" when "00100111",
X"0d65a2" when "00101000",
X"0ff654" when "00101001",
X"095a3e" when "00101010",
X"0f3694" when "00101011",
X"04054a" when "00101100",
X"033140" when "00101101",
X"000000" when "00101110",
X"0634d0" when "00101111",
X"061b1f" when "00110000",
X"0b8e48" when "00110001",
```

```
X"018b33" when "00110010",
X"003ec7" when "00110011",
X"06b4ae" when "00110100",
X"0d752b" when "00110101",
X"0f9ec1" when "00110110",
X"0eedbb" when "00110111",
X"09a3cd" when "00111000",
X"0b3a0c" when "00111001",
X"00f3b0" when "00111010",
X"0922b3" when "00111011",
X"097aba" when "00111100",
X"098724" when "00111101",
X"0093f7" when "00111110",
X"0d4a01" when "00111111",
X"07aa84" when "01000000",
X"032d4d" when "01000001",
X"02e7f3" when "01000010",
X"01767e" when "01000011",
X"06d974" when "01000100",
X"0d7a0b" when "01000101",
X"086621" when "01000110",
X"01bd0b" when "01000111",
X"009b05" when "01001000",
X"0765b7" when "01001001",
X"0408bc" when "01001010",
X"05ad06" when "01001011",
X"07470f" when "01001100",
X"09a05f" when "01001101",
X"0080e5" when "01001110",
X"08b643" when "01001111",
X"02c1cd" when "01010000",
X"0bb80a" when "01010001",
X"0a72e7" when "01010010",
X"0ec141" when "01010011",
X"039a52" when "01010100",
X"0346f5" when "01010101",
X"07db4b" when "01010110",
X"063218" when "01010111",
X"02f919" when "01011000",
X"054c6e" when "01011001",
X"0d1049" when "01011010",
X"0b02e6" when "01011011",
X"0f6c46" when "01011100",
X"0786ed" when "01011101",
X"0ea73e" when "01011110",
X"0b9641" when "01011111",
X"0eb33b" when "01100000",
X"07da09" when "01100001",
X"04d9cf" when "01100010",
X"075653" when "01100011",
X"033f26" when "01100100",
```

```
X"0a7ee8" when "01100101",
X"00da44" when "01100110",
X"012949" when "01100111",
X"093222" when "01101000",
X"0cf1d4" when "01101001",
X"06dad9" when "01101010",
X"0c37ac" when "01101011",
X"023a8a" when "01101100",
X"0b437d" when "01101101",
X"085f7f" when "01101110",
X"00754b" when "01101111",
X"09bc28" when "01110000",
X"07a8af" when "01110001",
X"09c5d8" when "01110010",
X"05d148" when "01110011",
X"0b4cd2" when "01110100",
X"08d6ec" when "01110101",
X"03c2e7" when "01110110",
X"00e5ff" when "01110111",
X"049e20" when "01111000",
X"0af0b7" when "01111001",
X"0dea2c" when "01111010",
X"02754f" when "01111011",
X"0981d7" when "01111100",
X"087813" when "01111101",
X"0225c8" when "01111110",
X"0bf1f9" when "01111111",
X"0ccaa5" when "10000000",
X"0cf1dc" when "10000001",
X"025b3e" when "10000010",
X"06caa4" when "10000011",
X"0b5ed6" when "10000100",
X"0198a6" when "10000101",
X"094c81" when "10000110",
X"021f33" when "10000111",
X"0fe6a0" when "10001000",
X"05b1b2" when "10001001",
X"0618e0" when "10001010",
X"0afd06" when "10001011",
X"0d139d" when "10001100",
X"07caa3" when "10001101",
X"0e957e" when "10001110",
X"0e32cc" when "10001111",
X"02931e" when "10010000",
X"0b5a8c" when "10010001",
X"0808e9" when "10010010",
X"084078" when "10010011",
X"0550f7" when "10010100",
X"0ef503" when "10010101",
X"0fa978" when "10010110",
X"07dc13" when "10010111",
```

```
X"06c456" when "10011000",
X"0271ca" when "10011001",
X"0c9210" when "10011010",
X"01f7e6" when "10011011",
X"08945b" when "10011100",
X"0a1fb6" when "10011101",
X"0bb4ed" when "10011110",
X"0b2ac8" when "10011111",
X"09f5f9" when "10100000",
X"0c4d75" when "10100001",
X"03a80d" when "10100010",
X"09cf07" when "10100011",
X"0ab8d7" when "10100100",
X"0926d2" when "10100101",
X"01fae5" when "10100110",
X"0e20a5" when "10100111",
X"0d0cbf" when "10101000",
X"021910" when "10101001",
X"0e18f4" when "10101010",
X"021785" when "10101011",
X"0a9d41" when "10101100",
X"0da50a" when "10101101",
X"0e90e5" when "10101110",
X"01ae1d" when "10101111",
X"0bb533" when "10110000",
X"0eb9ab" when "10110001",
X"04ebcf" when "10110010",
X"05b960" when "10110011",
X"07af9e" when "10110100",
X"036564" when "10110101",
X"021437" when "10110110",
X"0f6a68" when "10110111",
X"0d0fc2" when "10111000",
X"0de13e" when "10111001",
X"09a080" when "10111010",
X"010696" when "10111011",
X"052dd2" when "10111100",
X"0cac2d" when "10111101",
X"05285c" when "10111110",
X"0956c0" when "10111111",
X"0b7a04" when "11000000",
X"0aa13a" when "11000001",
X"0a20dc" when "11000010",
X"0ac204" when "11000011",
X"02eb90" when "11000100",
X"06a72e" when "11000101",
X"0c8ed2" when "11000110",
X"04fcef" when "11000111",
X"05454e" when "11001000",
X"0bcc5c" when "11001001",
X"073308" when "11001010",
```

```
X"00a98e" when "11001011",
X"076d82" when "11001100",
X"0e5522" when "11001101",
X"06a80b" when "11001110",
X"0f390d" when "11001111",
X"0c2a7c" when "11010000",
X"0077ca" when "11010001",
X"003d14" when "11010010",
X"0ed382" when "11010011",
X"0b29b7" when "11010100",
X"023469" when "11010101",
X"0f2daf" when "11010110",
X"09fe87" when "11010111",
X"0d4aff" when "11011000",
X"02eb5d" when "11011001",
X"0b29da" when "11011010",
X"0d2f3f" when "11011011",
X"08bd54" when "11011100",
X"010900" when "11011101",
X"09f614" when "11011110",
X"096867" when "11011111",
X"0f8f26" when "11100000",
X"0c513f" when "11100001",
X"033f1f" when "11100010",
X"027919" when "11100011",
X"09e77c" when "11100100",
X"0a09b1" when "11100101",
X"0990fe" when "11100110",
X"0964d4" when "11100111",
X"0a31b5" when "11101000",
X"0774bd" when "11101001",
X"0db4e5" when "11101010",
X"0ff17c" when "11101011",
X"0d197f" when "11101100",
X"06c48f" when "11101101",
X"0a8d9c" when "11101110",
X"094bda" when "11101111",
X"0948a5" when "11110000",
X"04e05a" when "11110001",
X"06de15" when "11110010",
X"0d2755" when "11110011",
X"036c49" when "11110100",
X"0b0aa4" when "11110101",
X"0039c3" when "11110110",
X"0eb16f" when "11110111",
X"0a1b3c" when "11111000",
X"06ab26" when "11111001",
X"0eb252" when "11111010",
X"0f25b5" when "11111011",
X"0acb0a" when "11111100",
X"0b32eb" when "11111101",
```

```
X"03aa6c" when "11111110",
X"04c2b8" when "11111111",
X"000000" when others;



end architecture;
```

2. fmPatch.vhd – creates the different audio patches

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

entity fmPatch is
     port(
          clk, reset_n : in std_logic;

          --1 when key is pressed and sounding, 0 when not sounding
          enable : in std_logic;

          --Allow up to 32 different patches
          mode   : in std_logic_vector(4 downto 0);

          --pitch (encoded as a carrier oscillator period)
          pitch  : in std_logic_vector(15 downto 0);

          --24-bit audio output for one voice
          fm_out : out std_logic_vector(23 downto 0)

     );
end fmPatch;

architecture fmPrtl of fmPatch is

     -- FM operator declaration
     component fmOperator is
          port (
               clk : in std_logic;
               reset_n : in std_logic;
               mute    : in std_logic;
               -- Oscillator select:
               --   0 = sine
               --  1 = saw
               --  2 = square
               --  3 = white noise (random)
               osc_select   : in std_logic_vector(1 downto 0);

               -- Oscillator frequency
               osc_freq   : in unsigned(15 downto 0);
```

```
            -- FM modulator
            modulator  : in unsigned(23 downto 0);
            mod_factor : in unsigned(4 downto 0);

            -- Sample out
            fm_out   : out std_logic_vector(23 downto 0)
        );
    end component;

    --outputs of different patches
    --        patch 0 - 3 are basic single waveforms (sin, saw,
sqaure, random)
    signal patch0, patch1, patch2, patch3 : std_logic_vector(23
downto 0);

    --fmOperator outputs
    signal fmOp_out_0, fmOp_out_1, fmOp_out_2, fmOp_out_3 :
std_logic_vector(23 downto 0);
    signal modTemp : unsigned(31 downto 0);
begin

    -- construct patches
    process(clk) begin
        if reset_n = '0' then
            patch0 <= x"000000";
            patch1 <= x"000000";
            patch2 <= x"000000";
            patch3 <= x"000000";
        else
            patch0 <= fmOp_out_0; --sin
            patch1 <= fmOp_out_1; --saw
            patch2 <= fmOp_out_2; --sqaure
            patch3 <= fmOp_out_3; --random
        end if;
    end process;

    -- select patch to output
    with mode select fm_out <=
        patch0 when "00000",
        patch1 when "00001",
        patch2 when "00010",
        patch3 when "00011",
        x"000000" when others;

        modTemp <= unsigned(pitch) * 3;

    -- Instantiate some FM operators from which to construct patches

    --fmOp0-3 take no modulators, use as first operators (unless
want a loop)
```

```
fmOp0 : fmOperator port map(
     clk => clk,
     reset_n => reset_n,
     mute => not enable,

     osc_select => "00", -- SINE

     osc_freq => unsigned(pitch),
     modulator => (others => '0'), --unsigned(fmOp_out_1),
     mod_factor => "01000",

     fm_out => fmOp_out_0
);

fmOp1 : fmOperator port map(
     clk => clk,
     reset_n => reset_n,
     mute => not enable,

     osc_select => "01", -- SAW

     osc_freq => unsigned(pitch),
     modulator => (others => '0'),
     mod_factor => "00010",

     fm_out => fmOp_out_1
);

fmOp2 : fmOperator port map(
     clk => clk,
     reset_n => reset_n,
     mute => not enable,

     osc_select => "00", -- SQUARE

     osc_freq => unsigned(pitch),
     modulator => unsigned(fmOp_out_3),
     mod_factor => "01111",

     fm_out => fmOp_out_2
);

fmOp3 : fmOperator port map(
     clk => clk,
     reset_n => reset_n,
     mute => not enable,

     osc_select => "00",

     osc_freq => modTemp(15 downto 0),
     modulator => (others => '0'),
```

```
          mod_factor => "00010",

          fm_out => fmOp_out_3
     );


end architecture;
```

3. synthesizer.vhd – the top level module for the synthesizer files

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

entity synthesizer is
     port(
          clk, reset_n : in std_logic;

          --FM synthesis inputs (from NIOS processor via Avalon bus)
          fm_dat_1, fm_dat_2, fm_dat_3, fm_dat_4, fm_dat_5 : in
std_logic_vector(15 downto 0);
          fm_en_1,  fm_en_2,  fm_en_3,  fm_en_4,  fm_en_5  : in
std_logic;

          --Select output (synthesizer or vocoder)
          select_out : std_logic;

          -- Switches
          SW  : in std_logic_vector(17 downto 0);

          -- I2C bus (from pins)
          I2C_SDAT : inout std_logic; -- I2C Data
          I2C_SCLK : out std_logic;   -- I2C Clock

          --Audio CODEC (from pins)
          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
     );
end synthesizer;

architecture srtl of synthesizer is
```

```
     --FM Patch declaration
     component fmPatch is
          port(
               clk, reset_n : in std_logic;

               --1 when key is pressed and sounding, 0 when not
sounding
               enable : in std_logic;

               --Allow up to 32 different patches
               mode   : in std_logic_vector(4 downto 0);

               --pitch (encoded as a carrier oscillator period)
               pitch  : in std_logic_vector(15 downto 0);

               --24-bit audio output for one voice
               fm_out : out std_logic_vector(23 downto 0)
          );
     end component;

     --Audio CODEC interface declaration
     component de2_wm8731_audio is
          port(
               clk: in std_logic; --Audio CODEC chip clock (AUD_XCK)
               reset_n : in std_logic;
               data_in : in std_logic_vector(23 downto 0); --Sample
data out
               data_out: out std_logic_vector(23 downto 0); --
Microphone sample data
               -- Audio interface signals
               AUD_ADCLRCK  : out std_logic;       --    Audio CODEC
ADC LR Clock
               AUD_ADCDAT   : in  std_logic;       --    Audio CODEC
ADC Data
               AUD_DACLRCK  : out std_logic;       --    Audio CODEC
DAC LR Clock
               AUD_DACDAT   : out std_logic;       --    Audio CODEC
DAC Data
               AUD_BCLK     : inout std_logic      --    Audio CODEC
Bit-Stream Clock
          );
     end component;

     --Audio CODEC configurator (drives i2C on reset)
     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;

     -- Vocoder declaration
     component bpfilterbank_imp is
          port ( Clk : in std_logic;
               input : in signed (31 downto 0);
               output : out signed(31 downto 0)
          );
               --   outp : out signed(0 downto 0)); --TEST
     end component;


     --signals
     signal audio_clock  : unsigned(1 downto 0) := "00";

     signal out_sample   : std_logic_vector(23 downto 0);
     signal in_sample    : std_logic_vector(23 downto 0);

     signal fm_sample_out : std_logic_vector(23 downto 0);
     signal vocoder_out   : std_logic_vector(23 downto 0);
     signal voc_tmp_out   : signed(31 downto 0);

     signal fm_out_1 : std_logic_vector(23 downto 0);
     signal fm_out_2 : std_logic_vector(23 downto 0);
     signal fm_out_3 : std_logic_vector(23 downto 0);
     signal fm_out_4 : std_logic_vector(23 downto 0);
     signal fm_out_5 : std_logic_vector(23 downto 0);
begin

     process (clk) begin
          -- FM synthesizer output is sum of fm voices
          if reset_n = '0' then
               fm_sample_out <= x"000000";
          else
               fm_sample_out <= std_logic_vector(signed(fm_out_1) +
signed(fm_out_2)
                                    + signed(fm_out_3) +
signed(fm_out_4) + signed(fm_out_5));
          end if;

     end process;

     -- Output can be straight from syntehsizer or vocoder
     out_sample <= fm_sample_out when select_out = '0' else
vocoder_out;

     -- Instantiate vocoder
     voc : bpfilterbank_imp port map (
          Clk => clk,
          input => signed(x"00" & in_sample),
          output => voc_tmp_out
```

```
);
vocoder_out <= std_logic_vector(voc_tmp_out(23 downto 0));


-- Instantiate an FM patch for each voice
fmPatch1 : fmPatch port map (
     clk => clk,
     reset_n => reset_n,
     enable => fm_en_1,
     mode => SW(17 downto 13),
     pitch  => fm_dat_1,
     fm_out => fm_out_1
);
fmPatch2 : fmPatch port map (
     clk => clk,
     reset_n => reset_n,
     enable => fm_en_2,
     mode => SW(17 downto 13),
     pitch  => fm_dat_2,
     fm_out => fm_out_2
);
fmPatch3 : fmPatch port map (
     clk => clk,
     reset_n => reset_n,
     enable => fm_en_3,
     mode => SW(17 downto 13),
     pitch  => fm_dat_3,
     fm_out => fm_out_3
);
fmPatch4 : fmPatch port map (
     clk => clk,
     reset_n => reset_n,
     enable => fm_en_4,
     mode => SW(17 downto 13),
     pitch  => fm_dat_4,
     fm_out => fm_out_4
);
fmPatch5 : fmPatch port map (
     clk => clk,
     reset_n => reset_n,
     enable => fm_en_5,
     mode => SW(17 downto 13),
     pitch  => fm_dat_5,
     fm_out => fm_out_5
);

--generate audio clock for audio codec driver
process (clk)
     begin
          if rising_edge(clk) then
               audio_clock <= audio_clock + "1";
```

```
            end if;
    end process;

    AUD_XCK <= audio_clock(1);

    --audio codec configurator (sets audio codec to 24-bit audio @
48kHz)
    i2c : de2_i2c_av_config port map (
         iCLK     => clk,
         iRST_n   => '1',
         I2C_SCLK => I2C_SCLK,
         I2C_SDAT => I2C_SDAT
    );

    --audio codec driver (serializes out_sample and deserializes mic
into in_sample)
    V1: de2_wm8731_audio port map (
         clk => audio_clock(1),
         reset_n => '1',
         data_in => out_sample,

         data_out => in_sample,
         -- Audio interface signals
         AUD_ADCLRCK  => AUD_ADCLRCK,
         AUD_ADCDAT   => AUD_ADCDAT,
         AUD_DACLRCK  => AUD_DACLRCK,
         AUD_DACDAT   => AUD_DACDAT,
         AUD_BCLK     => AUD_BCLK
    );
end architecture;
```

4. Part IV: Vocoder VHDL Files

1. bpfilterbank_en.vhd – bandpass filters two signals together

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

entity bpfilter_en is
port (  a1 : in signed(31 downto 0);    -- Although 16 bits long,we
will only use 8 to store the
a2 : in signed(31 downto 0);       -- information regarding
coefficients
a3 : in signed(31 downto 0);
b1 : in signed(31 downto 0);
b2 : in signed(31 downto 0);
b3 : in signed(31 downto 0);
x0 : in signed(31 downto 0);  -- 16 bit long amplitude input comes
from the audio codec
x1 : in signed(31 downto 0);  -- Amplitude input delayed by 1 time
```

```
unit
x2 : in signed(31 downto 0);  -- Amplitude input delayed by 2 time
units
y1 : in signed(31 downto 0);       -- Ampltiude output delayed by 1
time unit
y2 : in signed(31 downto 0);       -- Amplitude output delayed by 2
time units
--inp : in signed(0 downto 0);
y0 : out signed(31 downto 0));  -- naturally to stay consistent, we
have a 16 bit output

end bpfilter_en;

architecture bpfilter_ar of bpfilter_en is

signal y0buffer : signed(63 downto 0);

begin
--y0buffer <= ((b1 * x0) + (b2 * x1) + (b3 * x2) - (a2 * y1) - (a3 *
y2)) / a1;
--y0 <= y0buffer(31 downto 0);
y0buffer <= ((b1 * x0) + (b2 * x1) + (b3 * x2) - (a2 * y1) - (a3 *
y2));
y0 <= y0buffer(44 downto 13);    -- Even though we are cutting a 64
bit signal to a 32 bit signal,
-- we are losing no valuable information because the filter
-- does not amplify the input.
end bpfilter_ar;
```

2. bpfilterbank_imp.vhd – top-level entity which instantiates and executes 10 bandpass filters

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

entity bpfilterbank_imp is
port ( clk : in std_logic;
   inmod : in signed (31 downto 0);
   outmod1 : out signed(31 downto 0);
   outmod2 : out signed(31 downto 0);
   outmod3 : out signed(31 downto 0);
   outmod4 : out signed(31 downto 0);
   outmod5 : out signed(31 downto 0);
   outmod6 : out signed(31 downto 0);
   outmod7 : out signed(31 downto 0);
   outmod8 : out signed(31 downto 0);
   outmod9 : out signed(31 downto 0);
   outmod10 : out signed(31 downto 0));
--   outp : out signed(0 downto 0)); --TEST
end bpfilterbank_imp;
```

```
architecture bpfilterbank_ar of bpfilterbank_imp is



component bpfilter_en
port (  a1 : in signed(31 downto 0);  -- Although 32 bits long,we
will only use 8 to store the
a2 : in signed(31 downto 0);  -- information regarding coefficients
a3 : in signed(31 downto 0);
b1 : in signed(31 downto 0);
b2 : in signed(31 downto 0);
b3 : in signed(31 downto 0);
x0 : in signed(31 downto 0);  -- 32 bit long amplitude input comes
from the audio codec
x1 : in signed(31 downto 0);  -- Amplitude input delayed by 1 time
unit
x2 : in signed(31 downto 0);  -- Amplitude input delayed by 2 time
units
y1 : in signed(31 downto 0);  -- Ampltiude output delayed by 1 time
unit
y2 : in signed(31 downto 0);  -- Amplitude output delayed by 2 time
units
--inp : in signed(0 downto 0);
y0 : out signed(31 downto 0));  -- naturally to stay consistent, we
have a 32 bit output
end component;


-- Bandpass filter coefficients
signal a1 : signed(31 downto 0);
signal b2 : signed(31 downto 0);

signal bpf1a2 : signed(31 downto 0);
signal bpf1a3 : signed(31 downto 0);
signal bpf1b1 : signed(31 downto 0);
signal bpf1b3 : signed(31 downto 0);

signal bpf2a2 : signed(31 downto 0);
signal bpf2a3 : signed(31 downto 0);
signal bpf2b1 : signed(31 downto 0);
signal bpf2b3 : signed(31 downto 0);

signal bpf3a2 : signed(31 downto 0);
signal bpf3a3 : signed(31 downto 0);
signal bpf3b1 : signed(31 downto 0);
signal bpf3b3 : signed(31 downto 0);

signal bpf4a2 : signed(31 downto 0);
signal bpf4a3 : signed(31 downto 0);
signal bpf4b1 : signed(31 downto 0);
signal bpf4b3 : signed(31 downto 0);
```

```
signal bpf5a2 : signed(31 downto 0);
signal bpf5a3 : signed(31 downto 0);
signal bpf5b1 : signed(31 downto 0);
signal bpf5b3 : signed(31 downto 0);

signal bpf6a2 : signed(31 downto 0);
signal bpf6a3 : signed(31 downto 0);
signal bpf6b1 : signed(31 downto 0);
signal bpf6b3 : signed(31 downto 0);

signal bpf7a2 : signed(31 downto 0);
signal bpf7a3 : signed(31 downto 0);
signal bpf7b1 : signed(31 downto 0);
signal bpf7b3 : signed(31 downto 0);

signal bpf8a2 : signed(31 downto 0);
signal bpf8a3 : signed(31 downto 0);
signal bpf8b1 : signed(31 downto 0);
signal bpf8b3 : signed(31 downto 0);

signal bpf9a2 : signed(31 downto 0);
signal bpf9a3 : signed(31 downto 0);
signal bpf9b1 : signed(31 downto 0);
signal bpf9b3 : signed(31 downto 0);

signal bpf10a2 : signed(31 downto 0);
signal bpf10a3 : signed(31 downto 0);
signal bpf10b1 : signed(31 downto 0);
signal bpf10b3 : signed(31 downto 0);

-- Input signals
signal inmod1 : signed(31 downto 0);
signal inmod2 : signed(31 downto 0);
--
--signal incar1 : signed(31 downto 0);
--signal incar2 : signed(31 downto 0);

signal outputbuffer : signed(63 downto 0);

-- Intermediate Output Signals
signal modbuffer1 : signed(31 downto 0);
signal modbuffer2 : signed(31 downto 0);
signal modbuffer3 : signed(31 downto 0);
signal modbuffer4 : signed(31 downto 0);
signal modbuffer5 : signed(31 downto 0);

signal modbuffer6 : signed(31 downto 0);
signal modbuffer7 : signed(31 downto 0);
signal modbuffer8 : signed(31 downto 0);
signal modbuffer9 : signed(31 downto 0);
```

```
signal modbuffer10 : signed(31 downto 0);

--signal carbuffer1 : signed(31 downto 0);
--signal carbuffer2 : signed(31 downto 0);
--signal carbuffer3 : signed(31 downto 0);
--signal carbuffer4 : signed(31 downto 0);
--signal carbuffer5 : signed(31 downto 0);
--
--signal carbuffer6 : signed(31 downto 0);
--signal carbuffer7 : signed(31 downto 0);
--signal carbuffer8 : signed(31 downto 0);
--signal carbuffer9 : signed(31 downto 0);
--signal carbuffer10 : signed(31 downto 0);

-- State signals of band pass filter banks
signal outmod1_1 : signed(31 downto 0);
signal outmod2_1 : signed(31 downto 0);
--signal outcar1_1 : signed(31 downto 0);
--signal outcar2_1 : signed(31 downto 0);

signal outmod1_2 : signed(31 downto 0);
signal outmod2_2 : signed(31 downto 0);
--signal outcar1_2 : signed(31 downto 0);
--signal outcar2_2 : signed(31 downto 0);

signal outmod1_3 : signed(31 downto 0);
signal outmod2_3 : signed(31 downto 0);
--signal outcar1_3 : signed(31 downto 0);
--signal outcar2_3 : signed(31 downto 0);

signal outmod1_4 : signed(31 downto 0);
signal outmod2_4 : signed(31 downto 0);
--signal outcar1_4 : signed(31 downto 0);
--signal outcar2_4 : signed(31 downto 0);

signal outmod1_5 : signed(31 downto 0);
signal outmod2_5 : signed(31 downto 0);
--signal outcar1_5 : signed(31 downto 0);
--signal outcar2_5 : signed(31 downto 0);

signal outmod1_6 : signed(31 downto 0);
signal outmod2_6 : signed(31 downto 0);
--signal outcar1_6 : signed(31 downto 0);
--signal outcar2_6 : signed(31 downto 0);

signal outmod1_7 : signed(31 downto 0);
signal outmod2_7 : signed(31 downto 0);
--signal outcar1_7 : signed(31 downto 0);
--signal outcar2_7 : signed(31 downto 0);

signal outmod1_8 : signed(31 downto 0);
```

```
signal outmod2_8 : signed(31 downto 0);
--signal outcar1_8 : signed(31 downto 0);
--signal outcar2_8 : signed(31 downto 0);

signal outmod1_9 : signed(31 downto 0);
signal outmod2_9 : signed(31 downto 0);
--signal outcar1_9 : signed(31 downto 0);
--signal outcar2_9 : signed(31 downto 0);

signal outmod1_10 : signed(31 downto 0);
signal outmod2_10 : signed(31 downto 0);
--signal outcar1_10 : signed(31 downto 0);
--signal outcar2_10 : signed(31 downto 0);

--signal outp1: signed(0 downto 0);
--signal inp: signed (0 downto 0):= "1";
--signal clockdivider: signed(1 downto 0):= "00";
begin


a1 <= to_signed (8192, 32);
b2 <= to_signed (0, 32);  -- Not really necessary, but helps account
for everything

-- centered at 111Hz, 40Hz bandwidth, 32000Hz
bpf1a2 <= to_signed (-16340, 32);
bpf1a3 <= to_signed (8149, 32);
bpf1b1 <= to_signed (21, 32);
bpf1b3 <= to_signed (-21, 32);

-- centered at 250Hz, 50Hz bandwidth, 32000Hz
bpf2a2 <= to_signed (-16322, 32);
bpf2a3 <= to_signed (8139, 32);
bpf2b1 <= to_signed (27, 32);
bpf2b3 <= to_signed (-27, 32);

-- centered at 354Hz, 60Hz bandwidth, 32000Hz
bpf3a2 <= to_signed (-16302, 32);
bpf3a3 <= to_signed (8128, 32);
bpf3b1 <= to_signed (32, 32);
bpf3b3 <= to_signed (-32, 32);

-- centered at 500Hz, 70Hz bandwidth, 32000Hz
bpf4a2 <= to_signed (-16274, 32);
bpf4a3 <= to_signed (8117, 32);
bpf4b1 <= to_signed (38, 32);
bpf4b3 <= to_signed (-38, 32);

-- centered at 707Hz, 80Hz bandwidth, 32000Hz
bpf5a2 <= to_signed (-16229, 32);
bpf5a3 <= to_signed (8107, 32);
```

```
bpf5b1 <= to_signed (43, 32);
bpf5b3 <= to_signed (-43, 32);

 -- centered at 1000Hz, 90Hz bandwidth, 32000Hz
bpf6a2 <= to_signed (-16149, 32);
bpf6a3 <= to_signed (8096, 32);
bpf6b1 <= to_signed (48, 32);
bpf6b3 <= to_signed (-48, 32);

 -- centered at 1414Hz, 150Hz bandwidth, 32000Hz
bpf7a2 <= to_signed (-15947, 32);
bpf7a3 <= to_signed (8033, 32);
bpf7b1 <= to_signed (79, 32);
bpf7b3 <= to_signed (-79, 32);

 -- centered at 2000Hz, 250Hz bandwidth, 32000Hz
bpf8a2 <= to_signed (-15571, 32);
bpf8a3 <= to_signed (7928, 32);
bpf8b1 <= to_signed (132, 32);
bpf8b3 <= to_signed (-132, 32);

 -- centered at 2828Hz, 500Hz bandwidth, 32000Hz
bpf9a2 <= to_signed (-14790, 32);
bpf9a3 <= to_signed (7673, 32);
bpf9b1 <= to_signed (260, 32);
bpf9b3 <= to_signed (-260, 32);

-- centered at 5187Hz, 1000Hz bandwidth, 32000Hz
bpf10a2 <= to_signed (-11966, 32);
bpf10a3 <= to_signed (7184, 32);
bpf10b1 <= to_signed (504, 32);
bpf10b3 <= to_signed (-504, 32);



--bpf1 : bpfilter_en port map -- centered at 250Hz, 40Hz bandwidth,
32000Hz
----(a1, bpf1a2, bpf1a3, bpf1b1, b2, bpf1b3, input, input1, input2,
output1, output2, inp, y0buffer1);
--(a1, bpf1a2, bpf1a3, bpf1b1, b2, bpf1b3, input, input1, input2,
output1, output2, y0buffer1);

-- Modulating voice signal filtering
bpf1 : bpfilter_en port map
(a1, bpf1a2, bpf1a3, bpf1b1, b2, bpf1b3, inmod, inmod1, inmod2,
outmod1_1, outmod2_1, modbuffer1);
bpf2 : bpfilter_en port map
(a1, bpf2a2, bpf2a3, bpf2b1, b2, bpf2b3, inmod, inmod1, inmod2,
outmod1_2, outmod2_2, modbuffer2);
bpf3 : bpfilter_en port map
(a1, bpf3a2, bpf3a3, bpf3b1, b2, bpf3b3, inmod, inmod1, inmod2,
```

```
outmod1_3, outmod2_3, modbuffer3);
bpf4 : bpfilter_en port map
(a1, bpf4a2, bpf4a3, bpf4b1, b2, bpf4b3, inmod, inmod1, inmod2,
outmod1_4, outmod2_4, modbuffer4);
bpf5 : bpfilter_en port map
(a1, bpf5a2, bpf5a3, bpf5b1, b2, bpf5b3, inmod, inmod1, inmod2,
outmod1_5, outmod2_5, modbuffer5);

bpf6 : bpfilter_en port map
(a1, bpf6a2, bpf6a3, bpf6b1, b2, bpf6b3, inmod, inmod1, inmod2,
outmod1_6, outmod2_6, modbuffer6);
bpf7 : bpfilter_en port map
(a1, bpf7a2, bpf7a3, bpf7b1, b2, bpf7b3, inmod, inmod1, inmod2,
outmod1_7, outmod2_7, modbuffer7);
bpf8 : bpfilter_en port map
(a1, bpf8a2, bpf8a3, bpf8b1, b2, bpf8b3, inmod, inmod1, inmod2,
outmod1_8, outmod2_8, modbuffer8);
bpf9 : bpfilter_en port map
(a1, bpf9a2, bpf9a3, bpf9b1, b2, bpf9b3, inmod, inmod1, inmod2,
outmod1_9, outmod2_9, modbuffer9);
bpf10 : bpfilter_en port map
(a1, bpf10a2, bpf10a3, bpf10b1, b2, bpf10b3, inmod, inmod1, inmod2,
outmod1_10, outmod2_10, modbuffer10);


process(clk)
begin
if rising_edge(clk) then
--inp <= clockdivider(1 downto 1);
inmod1 <= inmod;
inmod2 <= inmod1;

outmod1_1 <= modbuffer1;
outmod2_1 <= outmod1_1;

outmod1_2 <= modbuffer2;
outmod2_2 <= outmod1_2;

outmod1_3 <= modbuffer3;
outmod2_3 <= outmod1_3;

outmod1_4 <= modbuffer4;
outmod2_4 <= outmod1_4;

outmod1_5 <= modbuffer5;
outmod2_5 <= outmod1_5;

outmod1_6 <= modbuffer6;
outmod2_6 <= outmod1_6;

outmod1_7 <= modbuffer7;
```

```
outmod2_7 <= outmod1_7;

outmod1_8 <= modbuffer8;
outmod2_8 <= outmod1_8;

outmod1_9 <= modbuffer9;
outmod2_9 <= outmod1_9;

outmod1_10 <= modbuffer10;
outmod2_10 <= outmod1_10;

outmod1 <= modbuffer1;
outmod2 <= modbuffer2;
outmod3 <= modbuffer3;
outmod4 <= modbuffer4;
outmod5 <= modbuffer5;
outmod6 <= modbuffer6;
outmod7 <= modbuffer7;
outmod8 <= modbuffer8;
outmod9 <= modbuffer9;
outmod10 <= modbuffer10;


--output <= modbuffer1 + modbuffer2 + modbuffer3 + modbuffer4 +
modbuffer5 + modbuffer6 + modbuffer7 + modbuffer8 + modbuffer9 +
modbuffer10;   -- We cannot read outputs directly, so this is the
best solution

--outp <= clockdivider(1 downto 1);
--clockdivider <= clockdivider + "1";
end if;
end process;

end bpfilterbank_ar;
```

3. sq_lowpass_en.vhd – performs a low pass filter on two signals

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

entity sq_lowpass_en is
port (
x0 : in signed(31 downto 0);  -- 16 bit long amplitude input comes
from the audio codec
x1 : in signed(31 downto 0);  -- Amplitude input delayed by 1 time
unit
y1 : in signed(31 downto 0);       -- Ampltiude output delayed by 1
time unit
y0 : out signed(31 downto 0));  -- naturally to stay consistent, we
have a 16 bit output
```

```
end sq_lowpass_en;

architecture sq_lowpass_ar of sq_lowpass_en is

signal y0buffer : signed(63 downto 0);
signal x0buffer : signed(63 downto 0);

begin
x0buffer <= (x0 * x0);
y0buffer <= ((229 * x0buffer(63 downto 32)) + (229 * x1) - (-7733 *
y1));
y0 <= y0buffer(44 downto 13);    -- Even though we are cutting a 32
bit signal to a 16 bit signal,
-- we are losing no valuable information because the filter
-- does not amplify the input.
end sq_lowpass_ar;
```

4. sq_lowpass_imp.vhd – top level lowpass module instantiating and executing 10 lowpass filters

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

entity sq_lowpass_imp is
port (  clk : in std_logic;
input1 : in signed(31 downto 0);   -- 16 bit long amplitude input
comes from the audio codec
input2 : in signed(31 downto 0);
input3 : in signed(31 downto 0);
input4 : in signed(31 downto 0);
input5 : in signed(31 downto 0);
input6 : in signed(31 downto 0);
input7 : in signed(31 downto 0);
input8 : in signed(31 downto 0);
input9 : in signed(31 downto 0);
input10 : in signed(31 downto 0);
output1 : out signed(31 downto 0);
output2 : out signed(31 downto 0);
output3 : out signed(31 downto 0);
output4 : out signed(31 downto 0);
output5 : out signed(31 downto 0);
output6 : out signed(31 downto 0);
output7 : out signed(31 downto 0);
output8 : out signed(31 downto 0);
output9 : out signed(31 downto 0);
output10 : out signed(31 downto 0));  -- naturally to stay
consistent, we have a 16 bit output

end sq_lowpass_imp;
```

```
architecture sq_lowpass_ar of sq_lowpass_imp is

component sq_lowpass_en
port (
x0 : in signed(31 downto 0);  -- 16 bit long amplitude input comes
from the audio codec
x1 : in signed(31 downto 0);  -- Amplitude input delayed by 1 time
unit
y1 : in signed(31 downto 0);       -- Ampltiude output delayed by 1
time unit
y0 : out signed(31 downto 0));  -- naturally to stay consistent, we
have a 16 bit output
end component;

signal input2_1 : signed(31 downto 0);
signal output2_1 : signed(31 downto 0);
signal outputbuffer_1 : signed(31 downto 0);

signal input2_2 : signed(31 downto 0);
signal output2_2 : signed(31 downto 0);
signal outputbuffer_2 : signed(31 downto 0);

signal input2_3 : signed(31 downto 0);
signal output2_3 : signed(31 downto 0);
signal outputbuffer_3 : signed(31 downto 0);

signal input2_4 : signed(31 downto 0);
signal output2_4 : signed(31 downto 0);
signal outputbuffer_4 : signed(31 downto 0);

signal input2_5 : signed(31 downto 0);
signal output2_5 : signed(31 downto 0);
signal outputbuffer_5 : signed(31 downto 0);

signal input2_6 : signed(31 downto 0);
signal output2_6 : signed(31 downto 0);
signal outputbuffer_6 : signed(31 downto 0);

signal input2_7 : signed(31 downto 0);
signal output2_7 : signed(31 downto 0);
signal outputbuffer_7 : signed(31 downto 0);

signal input2_8 : signed(31 downto 0);
signal output2_8 : signed(31 downto 0);
signal outputbuffer_8 : signed(31 downto 0);

signal input2_9 : signed(31 downto 0);
signal output2_9 : signed(31 downto 0);
signal outputbuffer_9 : signed(31 downto 0);

signal input2_10 : signed(31 downto 0);
```

```
signal output2_10 : signed(31 downto 0);
signal outputbuffer_10 : signed(31 downto 0);

begin

lpfilter1 : sq_lowpass_en port map
(input1, input2_1, output2_1, outputbuffer_1);

lpfilter2 : sq_lowpass_en port map
(input2, input2_2, output2_2, outputbuffer_2);

lpfilter3 : sq_lowpass_en port map
(input3, input2_3, output2_3, outputbuffer_3);

lpfilter4 : sq_lowpass_en port map
(input4, input2_4, output2_4, outputbuffer_4);

lpfilter5 : sq_lowpass_en port map
(input5, input2_5, output2_5, outputbuffer_5);

lpfilter6 : sq_lowpass_en port map
(input6, input2_6, output2_6, outputbuffer_6);

lpfilter7 : sq_lowpass_en port map
(input7, input2_7, output2_7, outputbuffer_7);

lpfilter8 : sq_lowpass_en port map
(input8, input2_8, output2_8, outputbuffer_8);

lpfilter9 : sq_lowpass_en port map
(input9, input2_9, output2_9, outputbuffer_9);

lpfilter10 : sq_lowpass_en port map
(input10, input2_10, output2_10, outputbuffer_10);

process(clk)
begin
if rising_edge(clk) then
input2_1 <= input1;
output2_1 <= outputbuffer_1;
output1 <= outputbuffer_1;

input2_2 <= input2;
output2_2 <= outputbuffer_2;
output2 <= outputbuffer_2;

input2_3 <= input3;
output2_3 <= outputbuffer_3;
output3 <= outputbuffer_3;

input2_4 <= input4;
```

```
output2_4 <= outputbuffer_4;
output4 <= outputbuffer_4;

input2_5 <= input5;
output2_5 <= outputbuffer_5;
output5 <= outputbuffer_5;

input2_6 <= input6;
output2_6 <= outputbuffer_6;
output6 <= outputbuffer_6;

input2_7 <= input7;
output2_7 <= outputbuffer_7;
output7 <= outputbuffer_7;

input2_8 <= input8;
output2_8 <= outputbuffer_8;
output8 <= outputbuffer_8;

input2_9 <= input9;
output2_9 <= outputbuffer_9;
output9 <= outputbuffer_9;

input2_10 <= input10;
output2_10 <= outputbuffer_10;
output10 <= outputbuffer_10;


end if;
end process;

end sq_lowpass_ar;
```

Appendix C. MATLAB Code Listings

1. Bandpassgen.m -- Band Pass Generator

```
function [num,den] = bandpassgen(fc, b, fs)

if (2*fc > fs), error('fs must be atleast 2*fc'); end
wc = 2*pi*fc/fs;
B = 2*pi*b/fs;
beta = cos(wc);
poly = [1 -2/cos(B) 1];
root = roots(poly);

if (root(1) < 1 && root(1) > 0)
    alpha = root(1);
elseif (root(2) < 1 && root(2) > 0)
    alpha = root(2);
else
    error('couldnt find alpha')
end

k = (1 - alpha)/2;
num = [k 0 -k];
den = [1 -beta*(1+alpha) alpha];
```

2. Lowpassgen.m -- Low Pass Filter

```
function [num den] = lowpassgen(fc,fs)

wc = 2*pi*fc/fs;
alpha = (1 - sin(wc))/cos(wc);
k = (1 - alpha)/2;

num = [k k];
den = [1 -alpha];
```

3. Filter_fix.m – Filter function

```
% From adapted code by Author: U. Zoelzer
% Code can handle up to second order transfer functions

function y = filter_fix( num, den, x)

len = length(num);
len2 = length(den);

if(len ~= len2)
     error('num and den should be of the same array size!');
end

if (len == 1)
    b(1) = num(1);
    a(1) = den(1);
    b(2) = 0;
    a(2) = 0;
    b(3) = 0;
```

```
        a(3) = 0;
    elseif (len == 2)
        b(1) = num(1);
        a(1) = den(1);
        b(2) = num(2);
        a(2) = den(2);
        b(3) = 0;
        a(3) = 0;
    elseif (len == 3)
        b(1) = num(1);
        a(1) = den(1);
        b(2) = num(2);
        a(2) = den(2);
        b(3) = num(3);
        a(3) = den(3);
    end

    % Initialization of state variables
    xh1=0;
    xh2=0;
    yh1=0;
    yh2=0;

    % Input signal: unit impulse
    N=length(x);  % length of input signal
    %x(N)=0;
    %x(1)=1;

    % Sample-by-sample algorithm
    for n=1:N
    y_float=(b(1)*x(n) + b(2)*xh1 + b(3)*xh2 - a(2)*yh1 - a(3)*yh2)/a(1) + 1;
    y(n) = floor(y_float);
    xh2=xh1;
    xh1=x(n);
    yh2=yh1;
    yh1=y(n);
    end;
```

4. Channelvocoder.m – Channel Vocoder implementation

```
function output = channelvocoder( mod, car, fs)

[num1, den1] = bandpassgen(111,40,fs);
[num2, den2] = bandpassgen(250,50,fs);
[num3, den3] = bandpassgen(354,60,fs);
[num4, den4] = bandpassgen(500,70,fs);
[num5, den5] = bandpassgen(707,80,fs);
[num6, den6] = bandpassgen(1000,90,fs);
[num7, den7] = bandpassgen(1414,150,fs);
[num8, den8] = bandpassgen(2000,250,fs);
[num9, den9] = bandpassgen(2828,500,fs);
[num10, den10] = bandpassgen(5187,1000,fs);

[lpfnum, lpfden] = lowpassgen(440,fs);

a_mod = filter(num1, den1, mod);
b_mod = filter(num2, den2, mod);
c_mod = filter(num3, den3, mod);
```

```
d_mod = filter(num4, den4, mod);
e_mod = filter(num5, den5, mod);
f_mod = filter(num6, den6, mod);
g_mod = filter(num7, den7, mod);
h_mod = filter(num8, den8, mod);
i_mod = filter(num9, den9, mod);
j_mod = filter(num10, den10, mod);

a_car = filter(num1, den1, car);
b_car = filter(num2, den2, car);
c_car = filter(num3, den3, car);
d_car = filter(num4, den4, car);
e_car = filter(num5, den5, car);
f_car = filter(num6, den6, car);
g_car = filter(num7, den7, car);
h_car = filter(num8, den8, car);
i_car = filter(num9, den9, car);
j_car = filter(num10, den10, car);


% Get the energy of the signal filtered components
a_mod_sq = 100*a_mod.^2;
b_mod_sq = 100*b_mod.^2;
c_mod_sq = 100*c_mod.^2;
d_mod_sq = 100*d_mod.^2;
e_mod_sq = 100*e_mod.^2;
f_mod_sq = 100*f_mod.^2;
g_mod_sq = 100*g_mod.^2;
h_mod_sq = 100*h_mod.^2;
i_mod_sq = 100*i_mod.^2;
j_mod_sq = 100*j_mod.^2;


% Get the envelope of the signal using low pass filter
a_mod_dc = filter(lpfnum, lpfden, a_mod_sq);
b_mod_dc = filter(lpfnum, lpfden, b_mod_sq);
c_mod_dc = filter(lpfnum, lpfden, c_mod_sq);
d_mod_dc = filter(lpfnum, lpfden, d_mod_sq);
e_mod_dc = filter(lpfnum, lpfden, e_mod_sq);
f_mod_dc = filter(lpfnum, lpfden, f_mod_sq);
g_mod_dc = filter(lpfnum, lpfden, g_mod_sq);
h_mod_dc = filter(lpfnum, lpfden, h_mod_sq);
i_mod_dc = filter(lpfnum, lpfden, i_mod_sq);
j_mod_dc = filter(lpfnum, lpfden, j_mod_sq);


% Multiplying respective components of modulator and carrier
a_out = a_mod_dc.*a_car;
b_out = b_mod_dc.*b_car;
c_out = c_mod_dc.*c_car;
d_out = d_mod_dc.*d_car;
e_out = e_mod_dc.*e_car;
f_out = f_mod_dc.*f_car;
g_out = g_mod_dc.*g_car;
h_out = h_mod_dc.*h_car;
i_out = i_mod_dc.*i_car;
j_out = j_mod_dc.*j_car;
```

```
% Summing the outputs;
output = a_out + b_out + c_out + d_out + e_out + f_out + g_out + h_out + i_out +
j_out;
```

5. Bandpassgen_fix.m – Bandpass Filter Generator

```
function [num,den] = bandpassgen_fix(fc, b, fs)

if (2*fc > fs), error('fs must be atleast 2*fc'); end

wc = 2*pi*fc/fs;
B = 2*pi*b/fs;
beta = cos(wc);
poly = [1 -2/cos(B) 1];
root = roots(poly);

if (root(1) < 1 && root(1) > 0)
    alpha = root(1);
elseif (root(2) < 1 && root(2) > 0)
    alpha = root(2);
else
    error('couldnt find alpha')
end

k = (1 - alpha)/2;
num = round(10000 * [k 0 -k]);
den = round(10000 * [1 -beta*(1+alpha) alpha]);
```

6. Lowpassgen_fix.m – Lowpass Filter generator

```
function [num den] = lowpassgen_fix(fc,fs)

wc = 2*pi*fc/fs;
alpha = (1 - sin(wc))/cos(wc);
k = (1 - alpha)/2;

num = round(1000000*[k k]);
den = round(1000000*[1 -alpha]);
```

7. Channelvocoder_fixedpt.m – Channel Vocoder

```
function output = channelvocoder_fixedpt( mod, car, fs)

[num1, den1] = bandpassgen_fix(111,40,fs);
[num2, den2] = bandpassgen_fix(250,50,fs);
[num3, den3] = bandpassgen_fix(354,60,fs);
[num4, den4] = bandpassgen_fix(500,70,fs);
[num5, den5] = bandpassgen_fix(707,80,fs);
[num6, den6] = bandpassgen_fix(1000,90,fs);
[num7, den7] = bandpassgen_fix(1414,150,fs);
[num8, den8] = bandpassgen_fix(2000,250,fs);
[num9, den9] = bandpassgen_fix(2828,500,fs);
[num10, den10] = bandpassgen_fix(5187,1000,fs);

[lpfnum, lpfden] = lowpassgen_fix(440,fs);
```

```
mod = (mod/max(mod))*1000000;  %Normalizes the modulating signal to between 1
and 1000000
car = (car/max(car))*1000000;  %Normalizes the carrier signal to between 1 and
1000000

a_mod = filter_fix(num1, den1, mod);
b_mod = filter_fix(num2, den2, mod);
c_mod = filter_fix(num3, den3, mod);
d_mod = filter_fix(num4, den4, mod);
e_mod = filter_fix(num5, den5, mod);
f_mod = filter_fix(num6, den6, mod);
g_mod = filter_fix(num7, den7, mod);
h_mod = filter_fix(num8, den8, mod);
i_mod = filter_fix(num9, den9, mod);
j_mod = filter_fix(num10, den10, mod);

a_car = filter_fix(num1, den1, car);
b_car = filter_fix(num2, den2, car);
c_car = filter_fix(num3, den3, car);
d_car = filter_fix(num4, den4, car);
e_car = filter_fix(num5, den5, car);
f_car = filter_fix(num6, den6, car);
g_car = filter_fix(num7, den7, car);
h_car = filter_fix(num8, den8, car);
i_car = filter_fix(num9, den9, car);
j_car = filter_fix(num10, den10, car);


% Get the energy of the signal filtered components
a_mod_sq = 100*a_mod.^2;
b_mod_sq = 100*b_mod.^2;
c_mod_sq = 100*c_mod.^2;
d_mod_sq = 100*d_mod.^2;
e_mod_sq = 100*e_mod.^2;
f_mod_sq = 100*f_mod.^2;
g_mod_sq = 100*g_mod.^2;
h_mod_sq = 100*h_mod.^2;
i_mod_sq = 100*i_mod.^2;
j_mod_sq = 100*j_mod.^2;


% Get the envelope of the signal using low pass filter
a_mod_dc = filter_fix(lpfnum, lpfden, a_mod_sq);
b_mod_dc = filter_fix(lpfnum, lpfden, b_mod_sq);
c_mod_dc = filter_fix(lpfnum, lpfden, c_mod_sq);
d_mod_dc = filter_fix(lpfnum, lpfden, d_mod_sq);
e_mod_dc = filter_fix(lpfnum, lpfden, e_mod_sq);
f_mod_dc = filter_fix(lpfnum, lpfden, f_mod_sq);
g_mod_dc = filter_fix(lpfnum, lpfden, g_mod_sq);
h_mod_dc = filter_fix(lpfnum, lpfden, h_mod_sq);
i_mod_dc = filter_fix(lpfnum, lpfden, i_mod_sq);
j_mod_dc = filter_fix(lpfnum, lpfden, j_mod_sq);


% Multiplying respective components of modulator and carrier
a_out = a_mod_dc.*a_car;
b_out = b_mod_dc.*b_car;
c_out = c_mod_dc.*c_car;
```

```
d_out = d_mod_dc.*d_car;
e_out = e_mod_dc.*e_car;
f_out = f_mod_dc.*f_car;
g_out = g_mod_dc.*g_car;
h_out = h_mod_dc.*h_car;
i_out = i_mod_dc.*i_car;
j_out = j_mod_dc.*j_car;


% Summing the outputs;
output = a_out + b_out + c_out + d_out + e_out + f_out + g_out + h_out + i_out +
j_out;
```