Guitar Effects

March 22, 2007

Navarun Jagatpal
Fred Rassam
Young Jin Yoon
Elton Chung

TABLE OF CONTENTS

I. Introduction

It is common for electric guitar players to make use of 'pedals' in order to achieve certain effects such as 'distortion/overdrive', 'delay/echo', 'chorus', 'flanger' and 'pitch/phase shifter'. Typically these pedals make use of analog electronics to yield the desired effects. Our goal will be to use the Altera DE2 FPGA board as a sampler and DSP (Digital Signal Processor) to produce a few of these effects. The sampling will be done in real-time with mono audio input from a guitar connected to a vacuum tube preamp into the FPGA. The output will be played on a pair of speakers.

II. System Overview

The diagram fig 1 below is a top level block diagram of our guitar effects system.

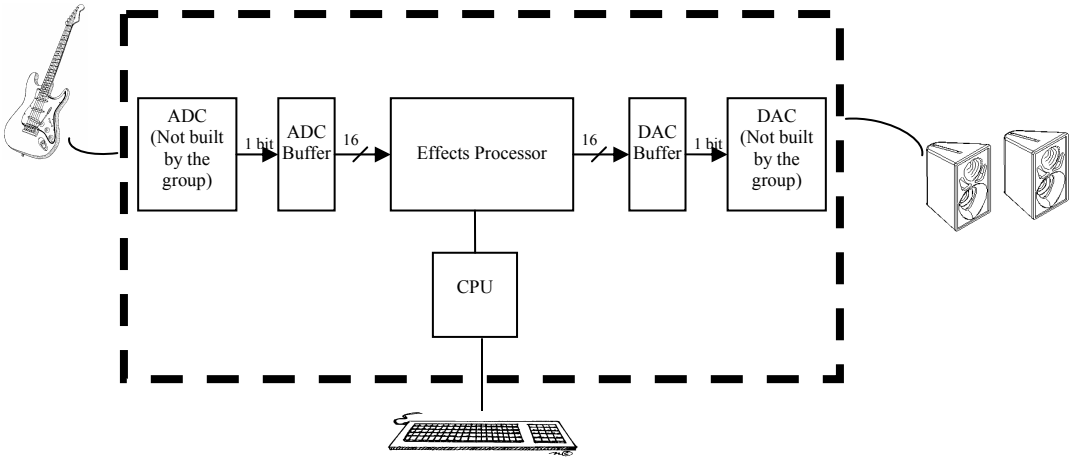

fig. 1 Top level diagram

The ADC(Audio Digital Converter) and DAC (Digital Audio Converter) will be done by the Wolfson WM8731 audio CODEC provided by the FPGA via the sound card. We will be designing a ADC buffer which will take the serial digital input received from the ADC and buffering it so our Effects Processor can modify the data to the desired effect before re-serializing the data out to the DAC through the DAC buffer. The CPU will be used to switch the effects, increase/decrease volume and increase/decrease the effect.

III. Hardware Requirements

I. Audio Codec

We will be using the Wolfson WM8731 audio CODEC for doing our analog digital conversions (ADC) and digital audio conversions (DAC). The CODEC

provides us with stereo and mono microphone level audio inputs as well as an array of programmable functions ranging form mute, volume control, bias voltage output, I2S, DSP, 16/20/24/32 bit Word Lengths and Master or Slave clocking mode. The CODEC samples the input with a range of frequency from 8kHz-96kHz

II. Effects Processor

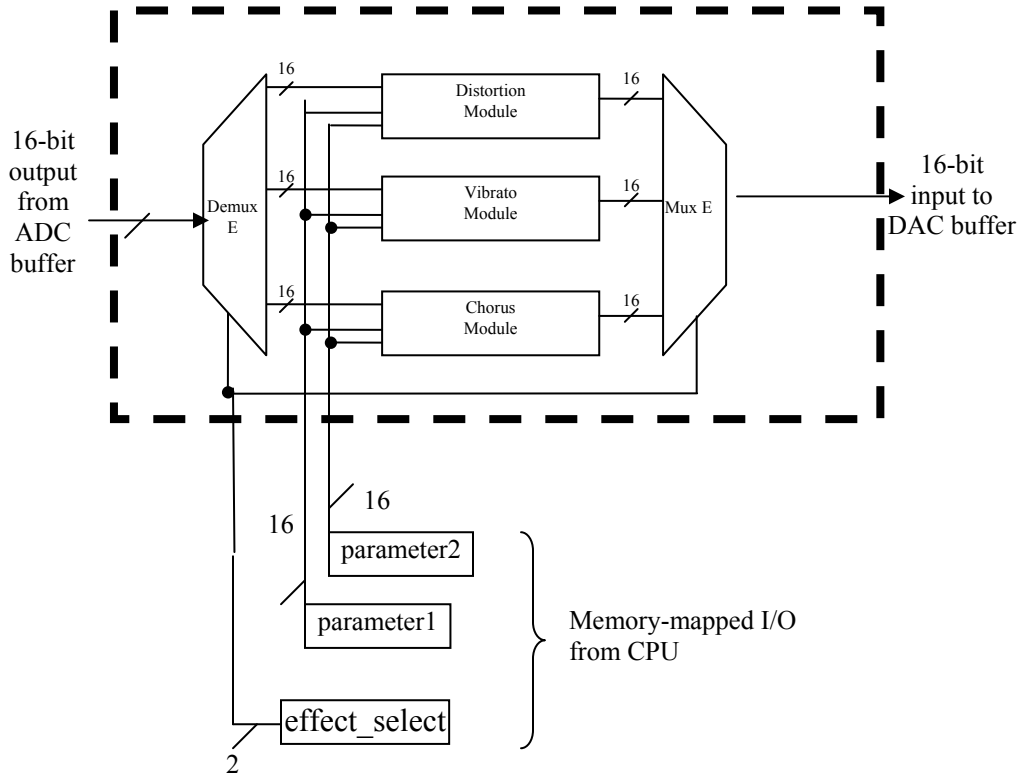

Fig. 2 Effects processor block diagram

The effects processor consists of a mux, demux and three effect modules. A 16-bit sample input feeds into the demux which supply inputs for the three separate effects, Distortion, Vibrato and Chorus, modules. The CPU will dictate which effect module to use and supply a gain and phase input parameter. The output will then be fed into a mux for aggregating the signals into a 16-bit buffer for the DAC to process.

1. Distortion Module

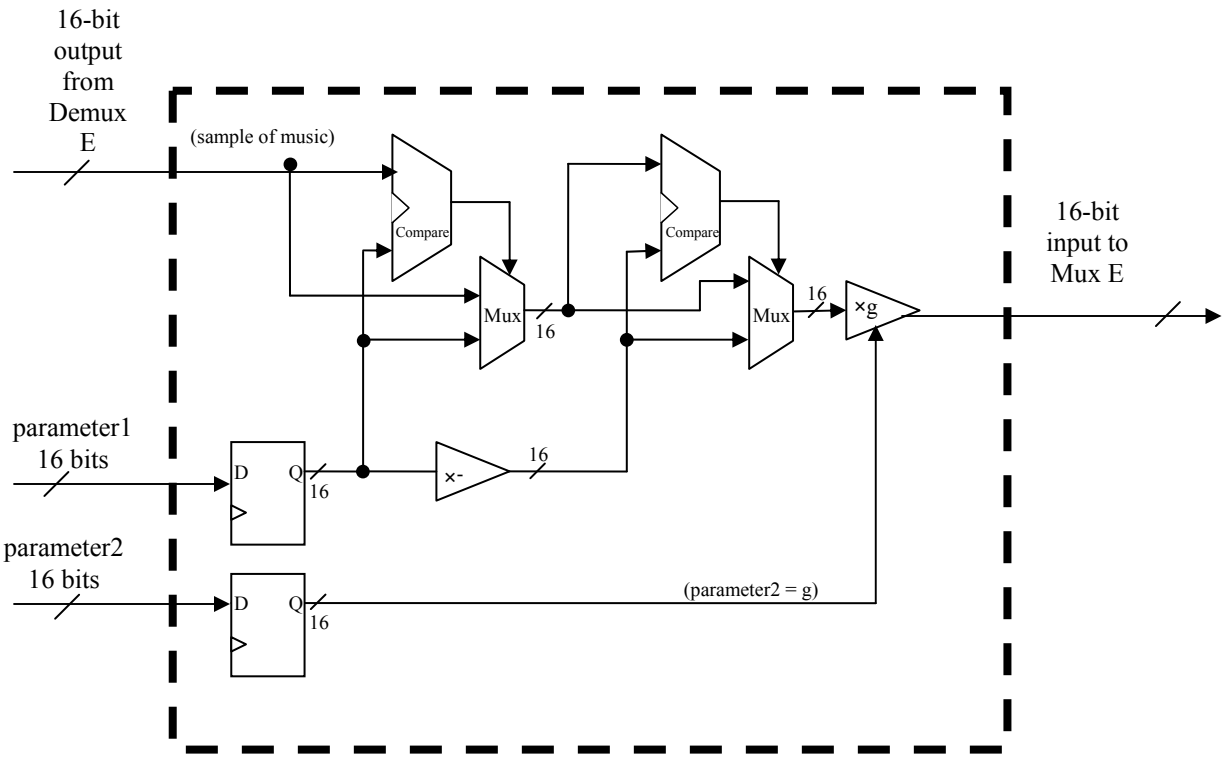

Fig. 3 Block Diagram of Distortion Module

The 16-bit input will be fed into two separate comparators. The first comparator will output a '1' if the sample is greater than 'N'. The second comparator outputs a '1' if the sample is less than -N. ('parameter1' = 'N'), 'N' being the clipping level

2. Vibrato Module

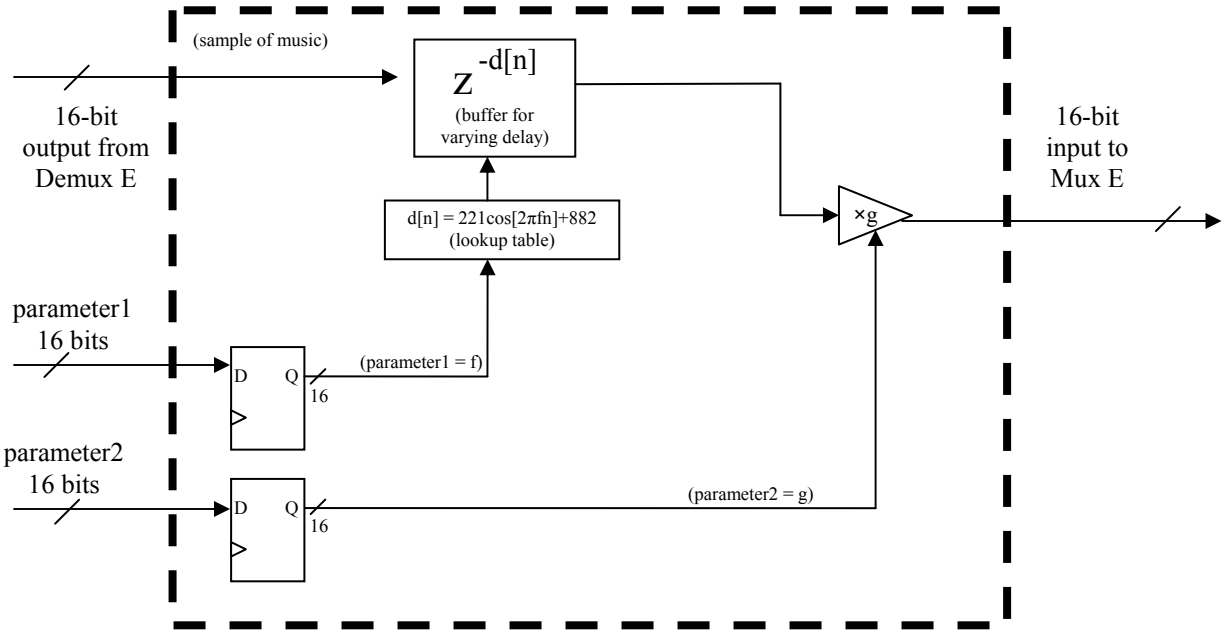

Fig 4. Block Diagram of Vibrato Module

The 16-bit signal will be re-evaluated with a complex cosine look-up table. A frequency parameter will be fed in from the CPU to help with the final calculations. The re-imaged signal will then be amplified by 'parameter2' which is also passed in through the CPU.

3. Chorus Module

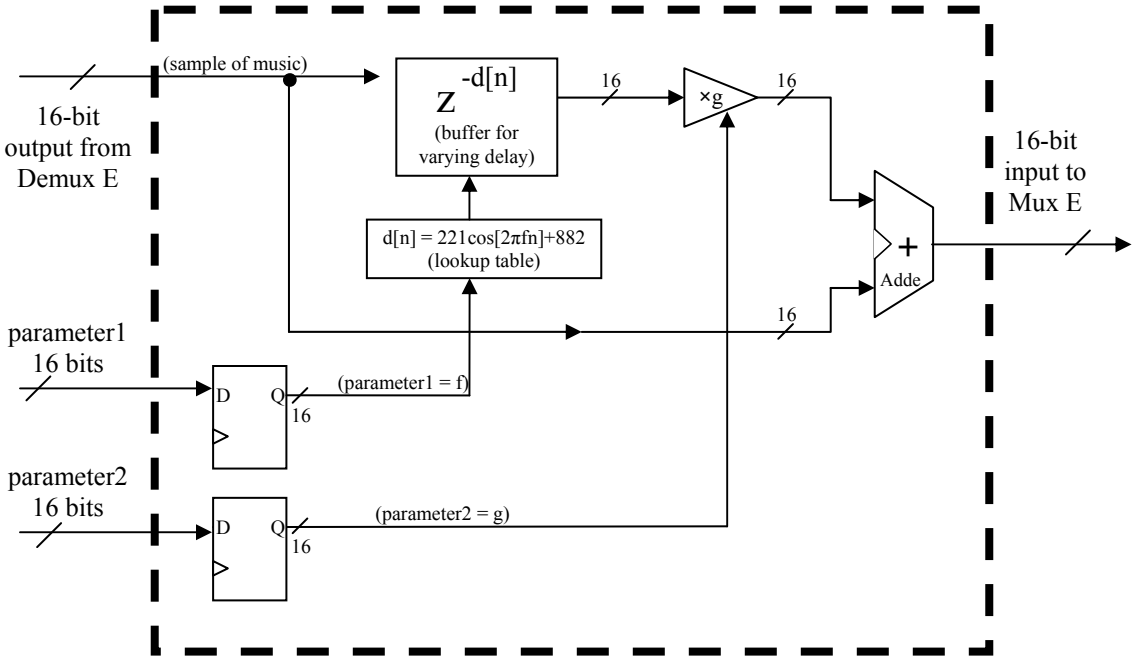

Fig. 5 Block Diagram of Chorus module

This module is similar to the Vibrato module except the resulting sample after re-imaging will be added with the original signal before sending it out to the mux.

IV. Software Requirements

The selection of which guitar effect to use will be done in software. This process will be mapped to different keys on a keyboard and the user will have the ability to choose which effects should take place.

APPENDIX

V. Appendix A

Verifying algorithm on C code

Because we need to verify the algorithm we used, we need to build a simple program in C to test our algorithm to implement functionality that we have. In our C program, basically it reads the sample from the file, applies the algorithm that we used, and then write a new sound file with applied sample. To read the sampling from file and write sample onto the file, we used libsndfile library from GCC. The programming interface that we used from libsndfile is shown on Table 1.

Name	Note
sf_open	Open the file to get / put the sample from them
sf_read_int	Read the sampling with integer type. If the file has different type for sampling, this function converts the sample on-the-fly.
sf_write_int	Write the sampling with integer type. If the file has different type fo r sampling, this function converts the sample on-the-fly.
sf_command	Give the various configuration to the libsndfile.

Currently we have tested the distortion algorithm with the code shown below. The results were as expected so now we are working on implementing Vibrato and Chorus.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sndfile.h>
#include <limits.h>

#define DEF_DIST_VAL (INT_MAX * 0.001)

int dist_val;


int distortion(int input);
void error(SNDFILE *reason);
void usage(char *filename);

int main(int argc, char *argv[])
{
      SNDFILE *src, *dst;
      SF_INFO src_info;
      char *src_name, *dst_name;
      char *tmp = NULL;

      sf_count_t src_cnt, dst_cnt;

      int src_data, dst_data;

      if(argc > 3 || argc < 1) usage(argv[0]);
      if(argc == 2) {
```

```
            dist_val = (int)DEF_DIST_VAL;
      } else {
            dist_val = strtol(argv[2], &tmp, 10);
            if(dist_val == 0 && tmp != NULL) usage(argv[0]);
      }
      printf("<DEBUG>: MAXIMUM - %d, VAL - %d\n",INT_MAX,dist_val);
      printf("<DEBUG>: before opening the src file\n");
            // open the source file
            src_name = argv[1];
            src = sf_open(src_name, SFM_READ, &src_info);
      printf("<DEBUG>: after opening the src file\n");
            // create the name of dest file
            dst_name = (char *)malloc(strlen(src_name)+4);
            strcpy(dst_name, src_name);
            strcat(dst_name, "_dst\0");
      printf("<DEBUG>: after changing the dst filename\n");

            // open the destination file
            dst = sf_open(dst_name, SFM_WRITE, &src_info);
      printf("<DEBUG>: after opening the dst file\n");

            // set float converting into integer
            sf_command (src, SFC_SET_SCALE_FLOAT_INT_READ, NULL,
SF_TRUE);
            sf_command (src, SFC_SET_CLIPPING, NULL, SF_TRUE);
            printf("<DEBUG>: after setting the command\n");

            while(1) {
                  src_cnt = sf_read_int (src, &src_data, 1);
                  if(src_cnt == 0) { sf_perror(src); break;}
                  else if(src_cnt < 0) error(src);
                  dst_data = distortion(src_data);
                  dst_cnt = sf_write_int(dst, &dst_data, 1);
                  if(dst_cnt <= 0) error(dst);
            }
      printf("<DEBUG>: Complete!\n");
}

int distortion(int input)
{
      if(input > dist_val) return (dist_val * 500);
      else if(input < -dist_val) return (-dist_val * 500);
      else return (input * 500);
}

void error(SNDFILE *reason)
{
      sf_perror(reason);
      exit(EXIT_FAILURE);
}

void usage(char *filename)
{
      fprintf(stdout,"Usage : %s <file_name>
[distortion_factor]\n",filename);
      exit(EXIT_SUCCESS);
}
```

Distortion Primer

A distortion is the alteration of the original shape (or other characteristic) of an object, image, sound, waveform or other form of information or representation. In the world of the guitar music, distortion is actively sought, evaluated, and appreciatively discussed in its endless favors. Comparison between distorted wave and original wave is in Figure 2.

Basic concept of implementation of that is amplifying sound so that the tops of the waveform clipped off in general analog circuit. However, because the sound values are represented as samples in digital sound processing, we can get more clear sound by clipping sampled value on the FPGA. The basic algorithm for implementing distortion is shown as Figure 3, and block diagram for our implementation in FPGA is in Figure 4.

```
if (sampled value > Maximum value for distortion)
        sampled value = Maximum value for distortion
else if (sampled value < - Maximum value for distortion)
        sampled value = - Maximum value for distortion
else
        sampled value = sampled value
```

Figure 1

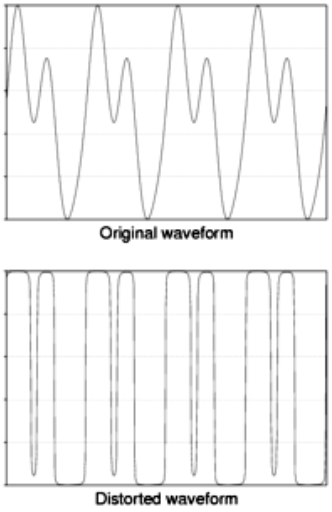

Figure 2

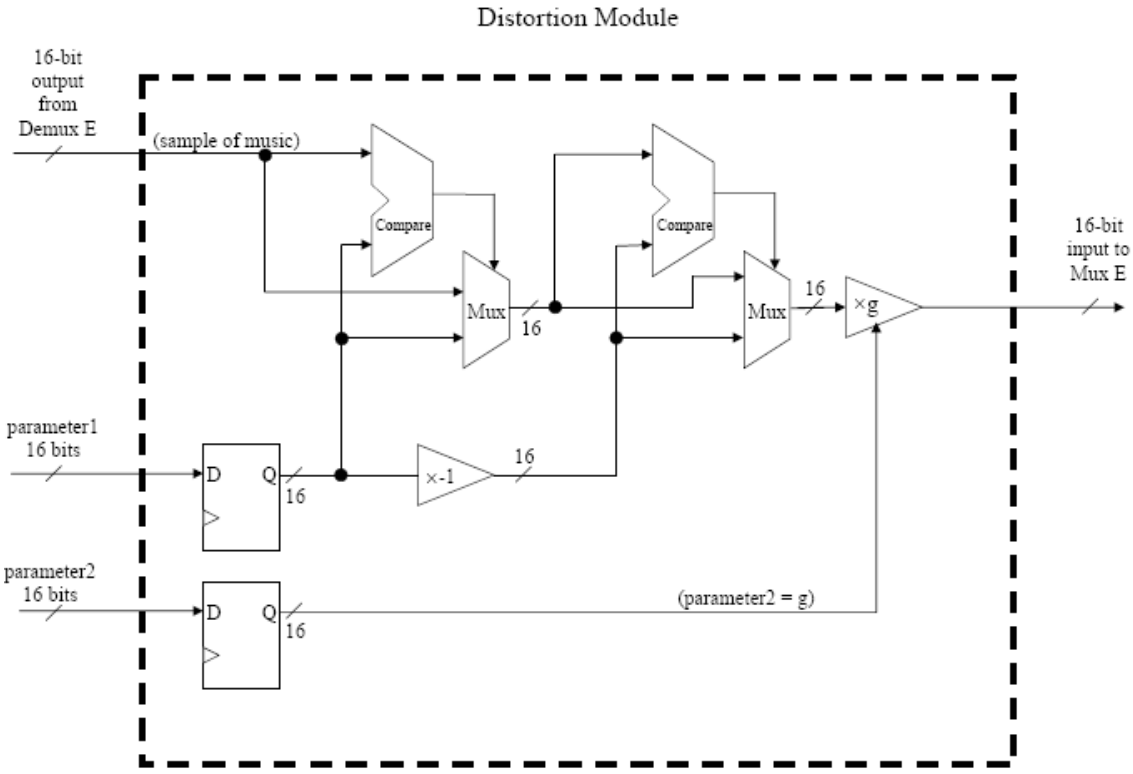

Figure 3

In block diagram, if the sample is greater than parameter1, the first comparator outputs 1 so that the Mux can take parameter1 rather than sampled data. For the same reason, the second comparator outputs a 1 if the sample is less than –parameter1, which chooses –parameter1 rather than sampled data. Because of this comparison, we can make clipped waveforms. After clipping the sound, it can be small so the human cannot catch it. So parameter2 is multiplied by the result of clipped samples.