Continuously Variable Slope Delta Modulation on a TI C5502 DSP Core

Real Time Digital Signal Processing - University of Nebraska

Daniel Shchur
62096122

Contents

1 Introduction

Continuing the previous projects, a Continuously Variable Slope Delta Modulation (CVSD) encoder and decoder was implemented in place of the mute button on the DSP board as a proof of concept audio compression technique. The project continues where the previous left off with no major changes to functionality of the FFT, display, and filters. The only thing changed is the mute button which now switches between using the filters or encoding and decoding audio samples through a CVSD algorithm with minor filtering applied.

2 CVSD

2.1 Approach

The CVSD algorithm is a method to encode audio data as a stream of single bits per sample, greatly saving on bandwidth at the expense of computational power and fidelity. The algorithm works by holding a reference point and comparing it with incoming audio samples. When the incoming audio is larger than or equal to the reference, the encoder returns a 1, otherwise it returns a 0. On a fixed delta, after every comparison, the delta is added to the reference if the bit is 1, otherwise it subtracts the delta. The implementation used in this case is continuously variable however, so a run of three encoded samples is kept in a buffer to vary the delta. When three samples are encoded as all 1s or all 0s, then the delta is increased since a "slope overload" is assumed to have occurred, otherwise, the delta is decreased. This allows for the reference to change is larger jumps if the signal continuously increases or decreases for a period of time. It also allows for a lower bit rate to achieve a much higher fidelity.

The decode process is the same but instead of encoding incoming samples as a bit, incoming bits are used in the same run of three and reference logic. The reference is the recreated sample in this case.

To aid in reducing distortions and bit rate, incoming samples and decoded samples can be filtered with a low pass filter.

2.2 Implementation

The implementation follows the algorithm very closely. Incoming 16bit samples are first filtered with a 22 pole low pass filter with a band pass frequency of 6000 Hz and a band stop frequency of 9000 Hz achieving -40 dB of attenuation at the stop frequency. This produces a fairly efficient and clean way of reducing distortions in the CVSD since most audio data doesn't go above 8000 Hz. Decoded 16bit samples are also filtered using the same filter before being sent to the audio codec.

The CVSD implementation itself uses a single 16bit integer as a delay line which bit shifts in encoded samples to use for the run of three logic. The delta is continuously decreased if no slope overload occurs and is not too small. On the other hand, if there occurs a run of 3 bits which are all 1s or 0s, then the delta is increased. Additional logic was attempted but produced worse results in decode fidelity.

When not overloading and only when the delta is larger than 1024, the delta is decreased by a summation of two divisions to emulate an intermediate division for implementation efficiency. The delta is bit shifted to the left by 1 to emulate a divide by 2. That value is then summed

with a delta which is bit shifted by 3 emulating a divide by 8. The whole process gives a multiplication by $\frac{5}{8}$.

On the other hand, when an overload does occur, the delta is increased by a subtraction of a multiplication and division. The delta is bit shifted to the left by 2 emulating a multiply by 2. That value is then subtracted by a delta which is bit shifted right by 2 emulating a divide by two. The combination nets a multiplication by $\frac{3}{2}$.

In every case, to prevent integer overflows and underflows, subtractions and additions are performed using saturated intrinsics.

For the purposes of this project, a filtered sample is encoded and immediately decoded, getting filtered again. In a real world use case of this algorithm, the decode and encoded would happen on separate audio streams and devices.

All code used for this portion of the project is appended in Section 5.

2.3 Efficiency

The implementation is fairly quick and memory efficient given care needs to be taken for each channel of data. The CVSD algorithm itself with encode and decode per sample takes about 245 cycles. At times it would go as low as 200 and as high as 800 depending on other BIOS processes interrupting the algorithm. When adding the filtering on the input and output, the cycles go up to around 480 per sample. This means that to process a single sample at 300 MHz, 0.00016% of the CPU is used. For a millisecond of stereo audio, 0.01536% is utilized. This leaves plenty of room for other processes, such as the FFT and display, to run.

Memory requirements are a little harder to calculate since it needs to be done implicitly using a difference of used bytes between a compiled version of the project with and without the CVSD. In this case, only the CVSD was measured since the filter algorithm is used elsewhere. The CVSD in its entirety takes up about 632 bytes of memory which includes separate variables for different channels of data. Had the CVSD been implemented for a single channel of audio, the memory usage would be cut down considerably as most operations and variables have to be duplicated for the other channel of audio.

3 Project Structure and Processes

3.0.1 BIOS Structure

The BIOS OS didn't change in structure from the previous project. The sample memory structure was simplified however to save considerably on memory requirements. Instead of encoding each sample as a structure of two variables to explicitly define the channel of audio, the samples were stripped to simply the audio data cutting memory requirements in half with respective mailboxes and queues changed accordingly to accommodate the decrease in memory requirements. To keep track of the channel, a variable is used as a flag which flips after every sample, implicitly defining the channel. This has the possible disadvantage of inverting channels of audio data given the current implementation. If audio data were to be deinterleaved, then the audio could be determined to be safe from that issue; however, that would require some more cycles since deinterleaved audio would have to be interleaved after the fact.

The order of tasks and other BIOS processes were left untouched.

3.1 FFT Screen

The FFT screen still performs at about the same frame rate as previously as very little overhead was added to the audio processing task with the CVSD algorithm. It's still around 17 frames a second.

4 Summary

Fast and efficient CVSD encode and decode was implemented and explored on the TI C5502 DSP core. The encoding allows for high bandwidth savings if the data were to be transferred over a network. The decode offers decent fidelity with some distortion, being acceptable for some forms of data. Previous portions of the project such as FFT, display, and user interface were able to be used without any changes despite the increase in computation costs. This was all performed through the use of threaded tasks utilizing the DSP BIOS framework.

5 Appendix

5.1 CVSD Encode

```
uint16_t encode_sample(int16_t sample, int16_t channel)
{
    static uint16_t delayL = 0, delayR = 0;
    static int16_t referenceL = 0, referenceR = 0;
    static uint16_t deltaL = 1024, deltaR = 1024;

    uint16_t bit = sample >= (channel ? referenceR : referenceL);

    if (channel == 1)
        delayR = (delayR << 1) | bit;
    else
        delayL = (delayL << 1) | bit;

    if (channel == 1)
        deltaR = var_delta(delayR, deltaR);
    else
        deltaL = var_delta(delayL, deltaL);

    if (bit == 1)
        if (channel == 1)
            referenceR = _sadd(referenceR, deltaR);
        else
            referenceL = _sadd(referenceL, deltaL);
    else
        if (channel == 1)
            referenceR = _ssub(referenceR, deltaR);
        else
            referenceL = _ssub(referenceL, deltaL);
    return bit;
```

```
}
```

5.2 CVSD Decode

```
int16_t decode_sample(uint16_t bit, int16_t channel)
{
    static uint16_t delayL = 0, delayR = 0;
    static int16_t referenceL = 0, referenceR = 0;
    static uint16_t deltaL = 1024, deltaR = 1024;

    if (channel == 1)
        delayR = (delayR << 1) | bit;
    else
        delayL = (delayL << 1) | bit;

    if (channel == 1)
        deltaR = var_delta(delayR, deltaR);
    else
        deltaL = var_delta(delayL, deltaL);

    if (bit == 1)
        if (channel == 1)
            referenceR = _sadd(referenceR, deltaR);
        else
            referenceL = _sadd(referenceL, deltaL);
    else
        if (channel == 1)
            referenceR = _ssub(referenceR, deltaR);
        else
            referenceL = _ssub(referenceL, deltaL);
    return channel ? referenceR : referenceL;
}
```

5.3 CVSD Run of 3

```
uint16_t var_delta(uint16_t delay, uint16_t delta)
{
    delay = (delay & 0x7);
    if (delay == 0x0 || delay == 0x7)
    {
        return _ssub(delta << 1, delta >> 2);
    }
    else if (delta > 0x3ff)
    {
        return _sadd(delta >> 1, delta >> 3);
    }
    return delta;
}
```

5.4 Audio Processing

```
myfir((const int16_t *) &samples[i],
    cvsdCoeff,
    &processedSample,
    lrFlag ? delayLineR : delayLineL,
    1,
    CVSD_COEFF_LEN
);
processedSample = decode_sample(encode_sample(processedSample, lrFlag), lrFlag);
myfir((const int16_t *) &processedSample,
    cvsdCoeff,
    &processedSample,
    lrFlag ? delayLineCvsdR : delayLineCvsdL,
    1,
    CVSD_COEFF_LEN
);
```