University of Victoria
Department of Software Engineering

SENG 440: Embedded Systems

Audio Compression using Mu Law
Final Report

Group 17

Name	Student Number	Email	Affiliation
Divya Chawla	[redacted]	[redacted]	Software Engineering
Purvika Dutt	[redacted]	[redacted]	Software Engineering

Submitted on: August 12th, 2020

Table of Content

List of Figures

List of Tables

1. Introduction

Audio compression is used in a large number of applications such as digital telephony, voice recording, digital music, etc. Pulse Code Modulation (PCM) is used to convert analog signals to digital signals, with higher voice quality and at a lower cost [3]. PCM consists of sampling, Quantization and Coding, where sampling refers to the determination of a signals amplitude at regular time intervals; Quantization refers to dividing the bandwidth of the system into quantization intervals; Coding is performed by converting the midpoint of each quantization level to a codeword [3]. The Quantization step introduces a small level of error because a lot of information deemed useless is discarded in this step. Uniform quantisation provides unneeded quality for large amplitude signals, and truncation effect on small amplitude signals, which prove to be inefficient because small amplitude signals contain the most information [3]. Thus, non-uniform quantisation is required which can be achieved by compressing the 14 bit sample to a 8 bit codeword, and then decompressing the 8 bit codeword back to 14 bit sample.

Audio compression is implemented using a logarithm like function, whereas audio expansion is implemented using exponential like function. Audio compression is achieved by increasing the quantization step for large signal levels, and keeping the quantization step the same for small signals. A large quantisation step would lower the number of bits to represent larger signals [1]. A Logarithm (non-uniform) PCM allows 8 bits per sample to represent the same range of values that would be encoded with 14 bits per sample uniform PCM, hence the compression ratio of 1.75:1 (original amount of information: compressed amount of information) [1].

For the purpose of this project, Mu Law logarithm function is used to achieve Audio Compression. A 11 second recording of our voice is saved as a .wav file, and this file is used to compress/decompress the speech signals using a software code written in C language. The original voice is then compared to the decompressed voice. The Software solution is optimized to increase the performance by reducing the processing time and the instruction count. A new instruction set is also proposed for mu law. At last, the C code is rewritten to use the proposed instruction set. The metrics used for the project includes processing time, instruction count, and latency of the system.

Following this section, a background of the project topic is discussed in Section 2 which explains the concepts used to implement Mu Law in C language. Section 3 explains the design and implementation of software and hardware solutions to our problem. Section 4 provides a discussion of any limitations, obstacles found during the course of the project. We will then

conclude the report, by providing our analyses and future work needed on the project, in Section 5. A reference and Appendix section is provided at the end of the report for further understanding of the report.

2. Background

The μ law quantizer function is represented by:

$$y = \operatorname{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)}$$

where the argument 0 <= |x| <= 1 and μ is a parameter which ranges from 0 (no compression) to 255 (maximum compression) [1].

Compression is achieved by using the logarithmic PCM that allows 8 bits per sample to represent the same range of values that would be encoded with 14 bits per sample using uniform PCM . Powers of 2's were used as threshold values since normal divisions are more expensive (3 times more expensive than bit shift operations) but divisions by 2 can be performed by using simple shift operations, which only take 1 cycle [1]. This also leads to us modifying our main μ-law quantizer to include log2 function rather than the natural logarithm function. Nowadays, in the digital era, the value of μ is chosen to be 255 to obtain maximum compression.

With μ = 255, and using log2 , the implied equation becomes:

$$y = \frac{1}{8}\log_2(1+255|x|)$$

The compression characteristics can be approximated by eight straight line segments called chords, where the slope of each chord is one-half the slope of the previous chord [1]. Each chord is divided into equally sized quantization intervals called steps [1]. The 8 bit codeword comprises a 1 sign bit concatenated with 3 bit chord concatenated with a 4 bit step [3]. The signal amplitude ranges and the corresponding chord slopes are displayed in Table 1.

Table 1: Mapping of Chord to Range

Chord Number	**Range**	**Chord Slope**
0	[0 - 31)	1
1	[31 - 95)	1/2

2	[95 - 223)	1/4
3	[223 - 479)	1/8
4	[479 - 991)	1/16
5	[991 - 2015)	1/32
6	[2015 - 4063)	1/64
7	[4063 - 8159)	1/128

In order to calculate the compressed 8 bit codeword from a 14 bit sample, first the input signal is converted to a sign-magnitude representation. A bias of 33 is added to the magnitude of input signal so that the threshold of the input signal: 31, 95, 223, 479, 991, 4063, and 8159, becomes power of 2 ie. 64, 128, 256, 512, 1024, 2048, 4096, 8192 . We then calculate the codeword from the resulting 14 bit signal. The 13 bits is represented as the magnitude, and the 14th bit is the sign bit (see Fig 1). The chord can be calculated by taking the position of the MSB of 1 in the magnitude (Using encoding table as shown in Fig 1). The four bits following the MSB of 1 gives us the step. And at last, we add one sign bit to the codeword, where '1' bit is added for positive samples and '0' bit is added for negative samples. The codeword is inverted before transmission [3].

Encoding examples:

0000001011010101 is encoded as **10100110**

(sign = 1, chord = 010, step = 0110, bits discarded = 101)

Biased Input Values in Signed-Magnitude Representation (sign bit, 13 magnitude bits)														Compressed Code Word (sign bit, 3 chord bits, 4 step bits)								
s	12	11	10	9	8	7	6	5	4	3	2	1	0	s	6	5	4	3	2	1	0	Chord
0/1	0	0	0	0	0	0	0	1	a	b	c	d	×	1/0	0	0	0	a	b	c	d	1st
0/1	0	0	0	0	0	0	1	a	b	c	d	×	×	1/0	0	0	1	a	b	c	d	2nd
0/1	0	0	0	0	0	1	a	b	c	d	×	×	×	1/0	0	1	0	a	b	c	d	3rd
0/1	0	0	0	0	1	a	b	c	d	×	×	×	×	1/0	0	1	1	a	b	c	d	4th
0/1	0	0	0	1	a	b	c	d	×	×	×	×	×	1/0	1	0	0	a	b	c	d	5th
0/1	0	0	1	a	b	c	d	×	×	×	×	×	×	1/0	1	0	1	a	b	c	d	6th
0/1	0	1	a	b	c	d	×	×	×	×	×	×	×	1/0	1	1	0	a	b	c	d	7th
0/1	1	a	b	c	d	×	×	×	×	×	×	×	×	1/0	1	1	1	a	b	c	d	8th

Fig 1. Encoding Table using Mu Law [1]

The compressed codeword is inverted back to the original codeword before starting with the decompression process. Decompression starts with an 8 bit compressed codeword , where the 1st bit is used to calculate the sign (1 for positive, and 0 for negative). The bits 2-4 denotes the chord in the compressed codeword, and we use this chord to find the magnitude of our decompressed sample (see fig 2). A bias of 33 is removed from the magnitude and is left shifted by 2 to obtain the decompressed data sample [3].

Decoding examples:

10100110 is decoded as **0000001011010 0**

(sign = 1, chord = 010, step = 0110)

Compressed Code Word (sign bit, 3 chord bits, 4 step bits)								Biased Output Values in Signed-Magnitude Representation (sign bit, 13 magnitude bits)														
s	6	5	4	3	2	1	0	s	12	11	10	9	8	7	6	5	4	3	2	1	0	Chord
1/0	0	0	0	a	b	c	d	0/1	0	0	0	0	0	0	0	1	a	b	c	d	1	1st
1/0	0	0	1	a	b	c	d	0/1	0	0	0	0	0	0	1	a	b	c	d	1	0	2nd
1/0	0	1	0	a	b	c	d	0/1	0	0	0	0	0	1	a	b	c	d	1	0	0	3rd
1/0	0	1	1	a	b	c	d	0/1	0	0	0	0	1	a	b	c	d	1	0	0	0	4th
1/0	1	0	0	a	b	c	d	0/1	0	0	0	1	a	b	c	d	1	0	0	0	0	5th
1/0	1	0	1	a	b	c	d	0/1	0	0	1	a	b	c	d	1	0	0	0	0	0	6th
1/0	1	1	0	a	b	c	d	0/1	0	1	a	b	c	d	1	0	0	0	0	0	0	7th
1/0	1	1	1	a	b	c	d	0/1	1	a	b	c	d	1	0	0	0	0	0	0	0	8th

Fig 2. Decoding Table using Mu Law [1]

3. Design

The Audio Compression program was written in C language. There were two approaches taken to achieve this. First, μ law was implemented as a Software approach using a look-up encoding/decoding table as discussed in the previous section. The C program was optimized to achieve a better performance. The next approach focused on presenting an architectural support for μ law operation by defining a new instruction set. Both approaches are discussed below in length.

3.1 Software Program Solution

Figure 12 in Appendix B shows the UML diagram for Audio Compression. The header of the .wav file is first parsed and the data samples are stored in an array. This array is then used in the compressDataSamples() function as shown in Figure 3.

```
void compressDataSamples() {
   printf("Compressing Data Samples.... \n");
   compressed_sample_data = calloc(num_samples, sizeof(char));
   if (compressed_sample_data == NULL) {
       printf("Could not allocate enough memory to store compressed data samples\n");
       return;
   }
   int i;
   for (i = 0; i < num_samples; i++) {
       int sample = (sample_data[i] >> 2);
       short sign = signum(sample);
       unsigned short magnitude = getMagnitude(sample) + 33;
       int codeword = codeword_compression(magnitude, sign);
       //Before transmission the µ-law codeword is inverted
       codeword = ~codeword;
       compressed_sample_data[i] = codeword;
   }
   printf("COMPLETED Compressing data samples \n\n");
}
short signum( int sample) {
   if( sample < 0)
       return( 0); /* sign is '0' for negative samples */
   else
       return( 1); /* sign is '1' for positive samples */
}
unsigned short getMagnitude( int sample) {
   if( sample < 0) {
       sample = -sample;
   }
   return (unsigned short) (sample);
}
```

Fig 3. C function to compress data samples

A sample is first right shifted by 2, and then converted to a sign-magnitude representation by calculating its sign and magnitude. A bias of 33 is added to the magnitude to convert it to a power of 2. Sign is easily calculated by assigning 0 to negative samples, and 1 to positive samples. Similarly, magnitude is calculated by changing the sign of a negative sample.The value of sign and magnitude is passed to another function which generates the compressed codeword (convert 14 bit to 8 bit) as shown in Figure 4. The chord and step is calculated based on the look up table described in Section 2 of the report (Figure 1). The compressed codeword is finally inverted, and stored in another array to keep track of compressed data samples.

```
int codeword_compression(unsigned short sample_magnitude, short sign) {
   int chord, step, codeword_tmp;
   if( sample_magnitude & (1 << 12)) {
       chord = 0x7;
       step = (sample_magnitude >> 8) & 0xF;
   }
   else if( sample_magnitude & (1 << 11)) {
       chord = 0x6;
       step = (sample_magnitude >> 7) & 0xF;
   }
   else if( sample_magnitude & (1 << 10)) {
       chord = 0x5;
       step = (sample_magnitude >> 6) & 0xF;
   }
   else if( sample_magnitude & (1 << 9)) {
       chord = 0x4;
       step = (sample_magnitude >> 5) & 0xF;
   }
   else if( sample_magnitude & (1 << 8)) {
       chord = 0x3;
       step = (sample_magnitude >> 4) & 0xF;
   }
   else if( sample_magnitude & (1 << 7)) {
       chord = 0x2;
       step = (sample_magnitude >> 3) & 0xF;
   }
   else if( sample_magnitude & (1 << 6)) {
       chord = 0x1;
       step = (sample_magnitude >> 2) & 0xF;
   }
```

```
    else {
        chord = 0x0;
        step = (sample_magnitude >> 1) & 0xF;
    }
    codeword_tmp = (sign << 7) | (chord << 4) | step;
    return ( (int)codeword_tmp);
}
```

Fig 4. C function to generate 8 bit compressed codeword

Figure 13 in Appendix B shows the UML diagram for Audio Decompression. The array that stores the compressed data samples is passed to the decompressDataSamples() function.

Decompression algorithm is used as explained in Section 2 of the report. A compressed sample is first inverted and stored in another variable called *codeword*. In order to calculate the sign, 1 sign bit is obtained by masking the codeword with 1000 0000 (0x80) and shifting the bit by 7. Then magnitude is calculated by sending the codeword to codeword_decompression() function. Chord is obtained by masking the codeword with 0111 0000 (0x70) and shifting the bit by 4. Step is obtained by masking the codeword with 0000 1111. Finally the magnitude is generated by combining the least significant 1, step and the most significant 1, as per look up table described in section 2 (Figure 2).

A bias of 33 is removed from the magnitude and is left shifted by 2 to obtain the decompressed data sample. This is stored in another array to keep track of decompressed data samples.

3.2 Optimization in Software Program Solution

3.2.1 Use of Local Variables

Local Variables provide better performance as compared to Global Variables because the compiler optimizes the local variables to be allocated from the registers, or from the cache [2]. Therefore, an attempt was made to eliminate the use of Global Variables by removing *num_samples* and *size_of_each_sample*.

3.2.2 Elimination of helper functions and variables

Helper functions such as signum and getMagnitude functions were removed and converted to inline functions to remove ‘jmp’ statements in the assembly file. Temporary variables used throughout the code were also removed to reduce instruction count of the overall program.

3.2.3 Use of Switch Statement

After experimenting with Switch and If else statements, Switch statements provided better execution time. Hence a switch statement was used to decompress the codeword instead of If...else statements.

3.2.4 Reversal of Chord Order

Since large amplitude signals are least likely to occur [1], so the order of chord checking was changed to check for the smaller chord values in if..else statements first during codeword compression.

3.2.5 Use of Loop Unrolling

Loop unrolling is used in all for loops throughout the code to improve the hardware performance by exploiting the parallelism between the loops [3]. It creates multiple copies of the loop body and adjusts the loop iteration counter accordingly [3]. Figure 5 shows the Loop Unrolling by a factor of 10.

```
for (i = 0; i < num_samples; i++) {
       #pragma HLS unroll factor=10
           fread(buffer2, size_of_each_sample, 1, fp);
           sample_data[i] = (buffer2[0]) | (buffer2[1] << 8);
       }
```

Fig 5. Loop Unrolling by a factor of 10

3.3 Hardware Solution

A new instruction set is created to calculate the compressed codeword, as shown in figure 6. This instruction is called *mulaw*, which takes in two input values - 14 bit data and mode operation (1 for compression), and returns a 8 bit codeword.

```
int codeword_compression(int sample) {
   //hardware instruction set
   const int mode_compression = 1;
   int codeword_tmp;
   __asm__ ("mulaw %0, %1, %2"
       : "=r" (codeword_tmp)
       : "r" (sample), "r" (mode_compression));
       return ( (int)codeword_tmp);
}
```

Fig 6. mulaw instruction set

A hardware unit is described in the next section in Figure 10. This instruction set can be further optimised by using the register variables, as shown in Figure 7.

```
int codeword_compression(register int sample) {
   //hardware instruction set
   register const int mode_compression = 1;
   register int codeword_tmp;
   __asm__ ("mulaw %0, %1, %2"
       : "=r" (codeword_tmp)
       : "r" (sample), "r" (mode_compression));
       return ( (int)codeword_tmp);
}
```

Fig 7. Optimized mulaw instruction set

4. Discussion

Our main software program written in C language parses a 10 sec audio wav file, processes and the samples into an array, and then performs compression/ decompression on the sample using Mu Law. A new .wav file is generated at the end of the program as a result of our decompressed sample data. When the two audio files were compared, there was some noise found in the new generated .wav file, but was still discernible.

The main C program was then optimised to get a better performance. When we were using the very first version of our code, our program had a total processing time of 460 ms. Fig 8 shows the execution of our main software code.

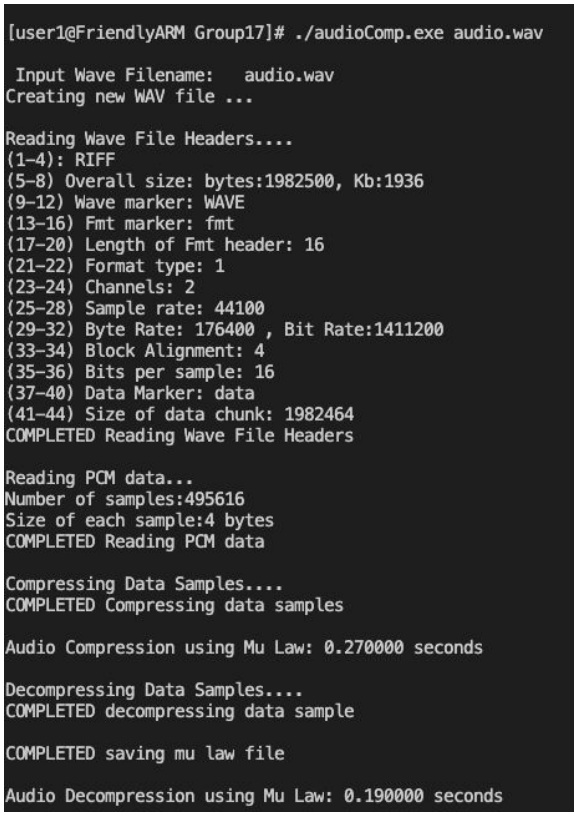

Fig 8. Main C code execution

However, after that we switched our if-else statements for the decompression code to switch statements. This reduced our assembly instruction count by a couple of lines and our processing time by around 2ms. We tried switching the compression code to use switch statements but that wasn't possible using C in a way that truly used the full functionality of switch rather than just using if-else statements inside a bigger switch block. The order of checking the chord during compression also improved the execution time by 5 ms.

Maximum Software Optimization was achieved by using loop unrolling. Loop unrolling allowed us to process many samples in parallel and that helped us improve our software processing time by around 20 ms. This was our final software optimization.

Table 2 lists the execution time of our Software code. The performance of our C code improved by 40.7% during compression and 15.8% during decompression, when we optimized our C program. The overall performance of the system was improved by 30.4%

Table 2: Execution time of C program

C Program	Compression duration (in ms)	Decompression duration (in ms)	Total execution time (in ms)

audioComp.c	270	190	460
opt.c	160	160	320

Table 3 lists the instruction count of C code. As we can see from Table 3, the instruction count improved by 137 lines of code. No significant improvement was observed when counting the instruction count of codeword_compression() and codeword_decompression() functions.

Table 3: Instruction Count of Software Solution

C Program	**Compression function (in # LOC)**	**Decompression function (in # LOC)**	**Total instruction count (in # LOC)**
audioComp.s	132	138	1560
opt.s	130	128	1423

After this, we checked the assembly code of both our original software solution and optimized software solution and realized we were limited in any further optimizations. However, we created an instruction set to support our hardware. Implementing this will improve the performance of our code massively. Already, we can see from figure 9(b) that a lot of load, store operations have been removed from our compression code and we are only left with the absolutely necessary instructions like move. We were able to reduce our instruction count by another 100 lines by using the hardware instruction set as seen in Table 4.

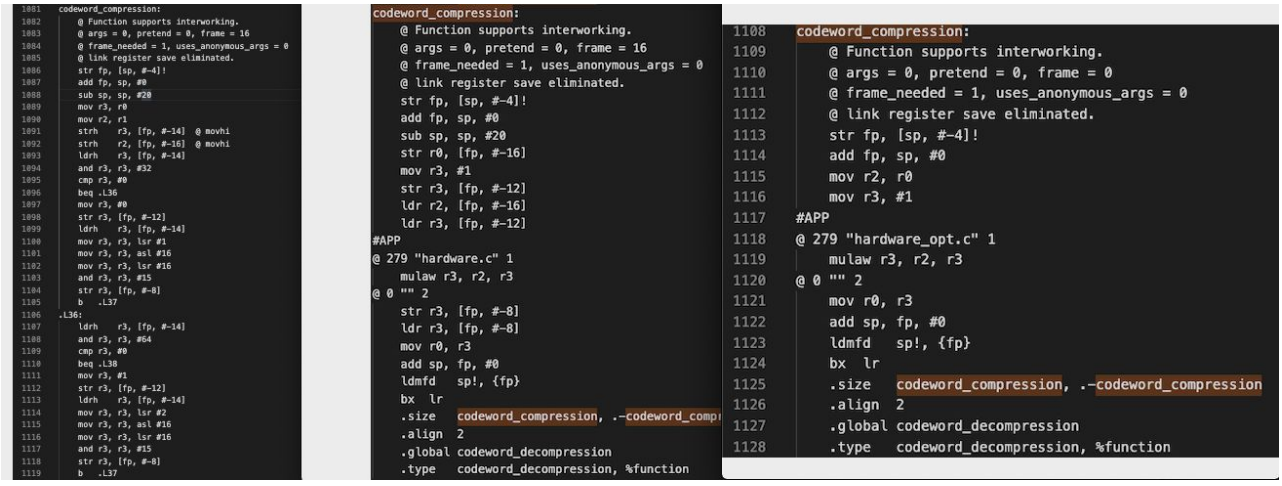

Fig 9 Assembly file in (a) opt.s (b) hardware.s (c) hardware_opt.s

We further improved our hardware performance by using specific registers for some variables which weren't being updated much to avoid having to set and unset different values, as discussed in Section 3.3. Since we only have 4 main values like sign, magnitude, chord and step, we were able to reduce our instruction count by a good 5-6 lines for our compression code which saves us a lot of processing time overall when we consider that the number of samples in our 10 second audio file is approximately 500,000.

Table 4: Instruction Count of Hardware Solution

C Program	**Compression function (in # LOC)**	**Total instruction count (in # LOC)**
hardware.s	23	1344
hardware_opt.s	17	1337

The latency of our proposed hardware instruction set is 3 cycles. Fig 10 shows the hardware support unit of mu law instruction. For the software solution, to get the codeword, we first need to get the sign which takes about 3-5 cycles (checking the if else condition and processing). Next, the software detects the chord, which includes 8 comparisons. There will be much more cycles because software solutions would involve masking, shifting and selection. In hardware, there is no comparison. We simply select the required bits out of 13 bits, which can be done by rewiring.

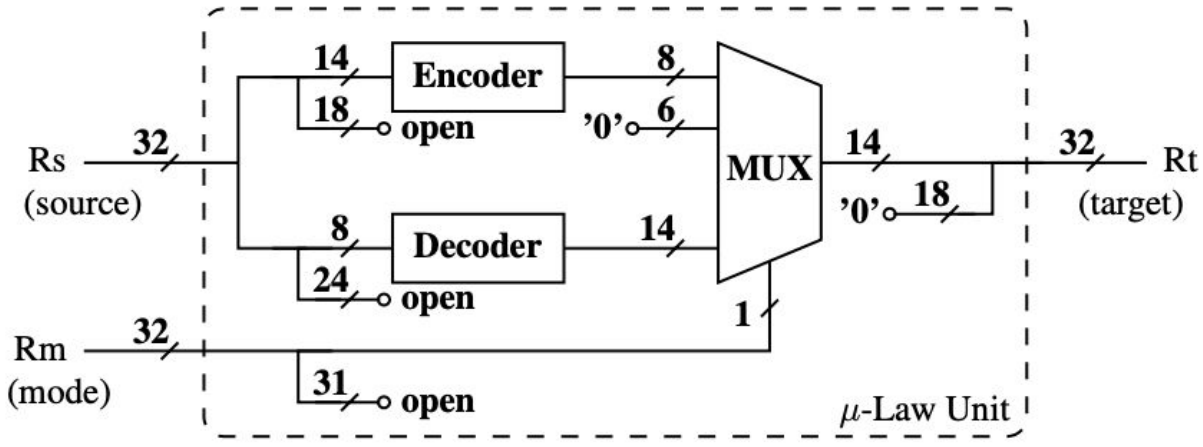

Fig 10. Hardware Support Unit for Mu Law instruction set

Some of the problems we encountered were with trying to install the Arm Virtual Machine on our local machines. We had many dependency issues with our MacBooks(Unix) and in spite of a lot of research, we couldn't figure out how to install all of the dependencies to run the VM on our machines. In the end, we decided to use Visual Studio Code to connect to the lab's ARM machines using a remote ssh at ugls.ece.uvic.ca. We compiled our program by first using our Mac in the beginning and then dealt with any compilation issues that showed up in the ARM machine by doing research.

One of the limitations we encountered was trying to transfer the files from our local machines to the ARM machines and vice versa. It was a long and tedious process to test any changes we made to our code(small or large) since we had to first transfer it to ugls, then build on the seng440 server and then execute on the ARM machine. Testing every little bit of code in this manner definitely added some challenges to our project.

Another limitation was collaborating in the beginning of the term due to Covid since people were still encouraged to stay away from everyone and not being able to work on the school lab computers. If we were able to collaborate on the lab computers, we would have been able to start our project a lot quicker since one of our biggest challenges was the lab environment set up. Thankfully, a little into the second month, we were able to start meeting up in person and collaborating well on this project. It also helped us feel more motivated to work on the project once we figured out the setup.

Meeting in person in the second month of the semester definitely speed things up as we were able to better figure out the issues, and work on our project. Also, scheduling meetings, and maintaining a project completion schedule helped us stay on track. The notes provided by the professor proved quite helpful to understand the problem, and utilize the helper methods to get us started on C programming.

5. Conclusion and Future Work

Audio compression is useful since it allows us to transmit information in an efficient manner since humans can only hear sound in a small range of frequencies. For this project, we have implemented audio compression using μ-law since it is one of the more commonly used methods in the USA and Japan. Our initial software implementation had a performance time of 460ms and after performing optimizations like removing helper functions, introducing switch statements, reversal of chord order, and loop unrolling, we had a performance time of 320 ms. Thus, our software performance improved by 30.43%. With the software implementation outlined in this report, the sample WAV file which was compressed and decompressed had some noise but the voice was still discernible. After we proposed a new instruction set for our hardware, we noticed a difference such that the load, store operations were reduced and only the more important move instructions were left.

In the future, we will try to do more research and find ways to reduce/remove the noise that has been introduced to the final sample. Other than that, we will also try to implement our instruction set in hardware since we have noticed that even a small improvement in our instruction count will reduce our latency and will significantly improve our overall performance since the number of lines reduced are multiplied with the number of samples.

6. References

[1] Lesson 104: Audio Compression, SENG 440 Embedded Systems, Mihai Sima. Accessed August 10th, 2020.

[2] L. Valencia, "Are global variables faster than local variables in C?", Stack Overflow, 2020. [Online]. Available: https://stackoverflow.com/questions/41147208/are-global-variables-faster-than-local-variables-in-c. [Accessed: 1- Aug- 2020].

[3] "Loop Pipelining and Loop Unrolling", Xilinx.com, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html. [Accessed: 20- Jul- 2020].

[4] Charles W. Brokish and Michele Lewis, “A-Law and mu-Law Companding Implementations Using the TMS320C54x”, Application Note SPRA163A, Texas Instruments, December 1997 [Online]. Available: https://www.ti.com/lit/an/spra163a/spra163a.pdf [Accessed: 5- Aug- 2020].

[5] "Parsing a WAV file in C", Truelogic Blog, 2020. [Online]. Available: http://truelogic.org/wordpress/2015/09/04/parsing-a-wav-file-in-c/. [Accessed: 12- June 2020].

[6] "Wav (RIFF) File Format Tutorial", Topherlee.com, 2020. [Online]. Available: http://www.topherlee.com/software/pcm-tut-wavformat.html. [Accessed: 27- June- 2020].

Appendix A - Optimised C Code

The following figure displays the code snippet for important functions from our optimized C code file - opt.c.

```
/*
   Compression
*/
void compression(long num_samples){
   start = clock();

   printf("Compressing Data Samples.... \n");
   compressed_sample_data = calloc(num_samples, sizeof(char));
   if (compressed_sample_data == NULL) {
       printf("Could not allocate enough memory to store compressed data samples\n");
       return;
   }
   int i;
   for (i = 0; i < num_samples; i++) {
   #pragma HLS unroll factor=10
       int sample = (sample_data[i] >> 2);
       short sign = sample >=0;
       unsigned short magnitude = (sample < 0 ? -sample : sample) + 33;
       //optimised
       compressed_sample_data[i] = ~(codeword_compression(magnitude, sign));
   }
   printf("COMPLETED Compressing data samples \n\n");

   stop = clock();
   //optimised
   printf("Audio Compression using Mu Law: %f seconds \n\n", (double) (stop - start) /
CLOCKS_PER_SEC);

}

/*
   Decompression
*/
void decompression(FILE *new_fp, long num_samples, long size_of_each_sample){
```

```
   start = clock();

   printf("Decompressing Data Samples....\n");
   int codeword;
   int i;
   for (i = 0; i < num_samples; i++) {
   #pragma HLS unroll factor=10
       //Before transmission the μ-law codeword is inverted
       codeword = ~(compressed_sample_data[i]);
       short sign = (codeword & 0x80) >> 7;
       unsigned short magnitude = (codeword_decompression(codeword) - 33);
       short sample = (short) (sign ? magnitude : -magnitude);
       sample_data[i] = sample << 2;
   }
   printf("COMPLETED decompressing data sample \n\n");

   stop = clock();
   saveFile(new_fp, num_samples, size_of_each_sample);

   //optimised
   printf("Audio Decompression using Mu Law: %f seconds\n\n", (double) (stop - start)
/ CLOCKS_PER_SEC);

}


int codeword_compression(unsigned short sample_magnitude, short sign) {
   int chord, step;

   if( sample_magnitude & (1 << 5)) {
       chord = 0x0;
       step = (sample_magnitude >> 1) & 0xF;
   }
   else if( sample_magnitude & (1 << 6)) {
       chord = 0x1;
       step = (sample_magnitude >> 2) & 0xF;
   }
   else if( sample_magnitude & (1 << 7)) {
       chord = 0x2;
```

```
        step = (sample_magnitude >> 3) & 0xF;
    }
    else if( sample_magnitude & (1 << 8)) {
        chord = 0x3;
        step = (sample_magnitude >> 4) & 0xF;
    }
    else if( sample_magnitude & (1 << 9)) {
        chord = 0x4;
        step = (sample_magnitude >> 5) & 0xF;
    }
    else if( sample_magnitude & (1 << 10)) {
        chord = 0x5;
        step = (sample_magnitude >> 6) & 0xF;
    }
    else if( sample_magnitude & (1 << 11)) {
        chord = 0x6;
        step = (sample_magnitude >> 7) & 0xF;
    }
    else {
        chord = 0x7;
        step = (sample_magnitude >> 8) & 0xF;
    }
    //optimised
    return ( (int)((sign << 7) | (chord << 4) | step));
}

unsigned short codeword_decompression(char codeword) {
    int chord = (codeword & 0x70) >> 4;
    int step = codeword & 0x0F;
    int msb = 1, lsb = 1;
    int magnitude;

    switch(chord) {
        case 0x0:
            magnitude = lsb | (step << 1) | (msb << 5);
            break;
        case 0x1:
            magnitude = (lsb << 1) | (step << 2) | (msb << 6);
            break;
```

```
        case 0x2:
            magnitude = (lsb << 2) | (step << 3) | (msb << 7);
            break;
        case 0x3:
            magnitude = (lsb << 3) | (step << 4) | (msb << 8);
            break;
        case 0x4:
            magnitude = (lsb << 4) | (step << 5) | (msb << 9);
            break;
        case 0x5:
            magnitude = (lsb << 5) | (step << 6) | (msb << 10);
            break;
        case 0x6:
            magnitude = (lsb << 6) | (step << 7) | (msb << 11);
            break;
        case 0x7:
            magnitude = (lsb << 7) | (step << 8) | (msb << 12);
            break;
    }
    return (unsigned short) magnitude;
}
```

Fig 11. Code Snippet of important functions from opt.c

Appendix B - UML Diagrams

Figure 12 and 11 displays the UML diagram of our main audio compression/ decompression functions from our audioComp.c file

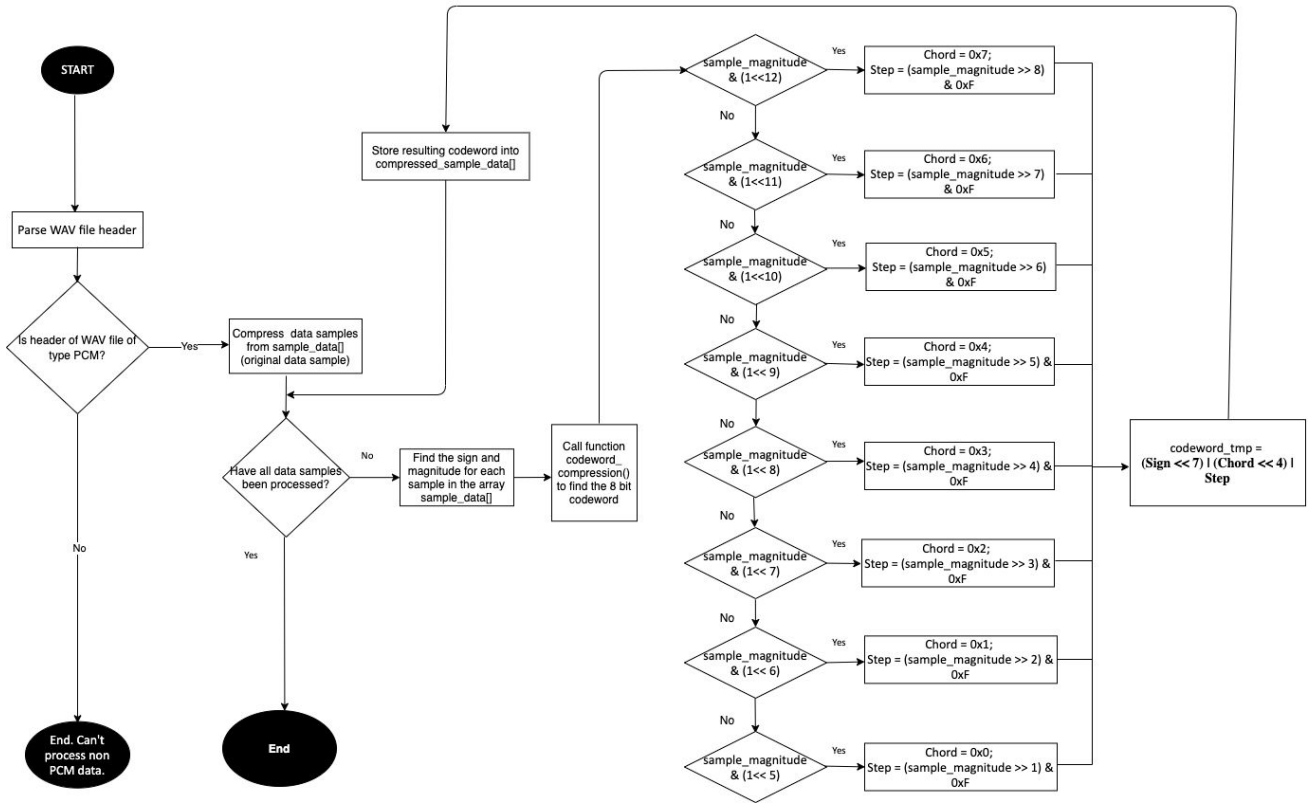

Fig 12. UML Diagram for Compression

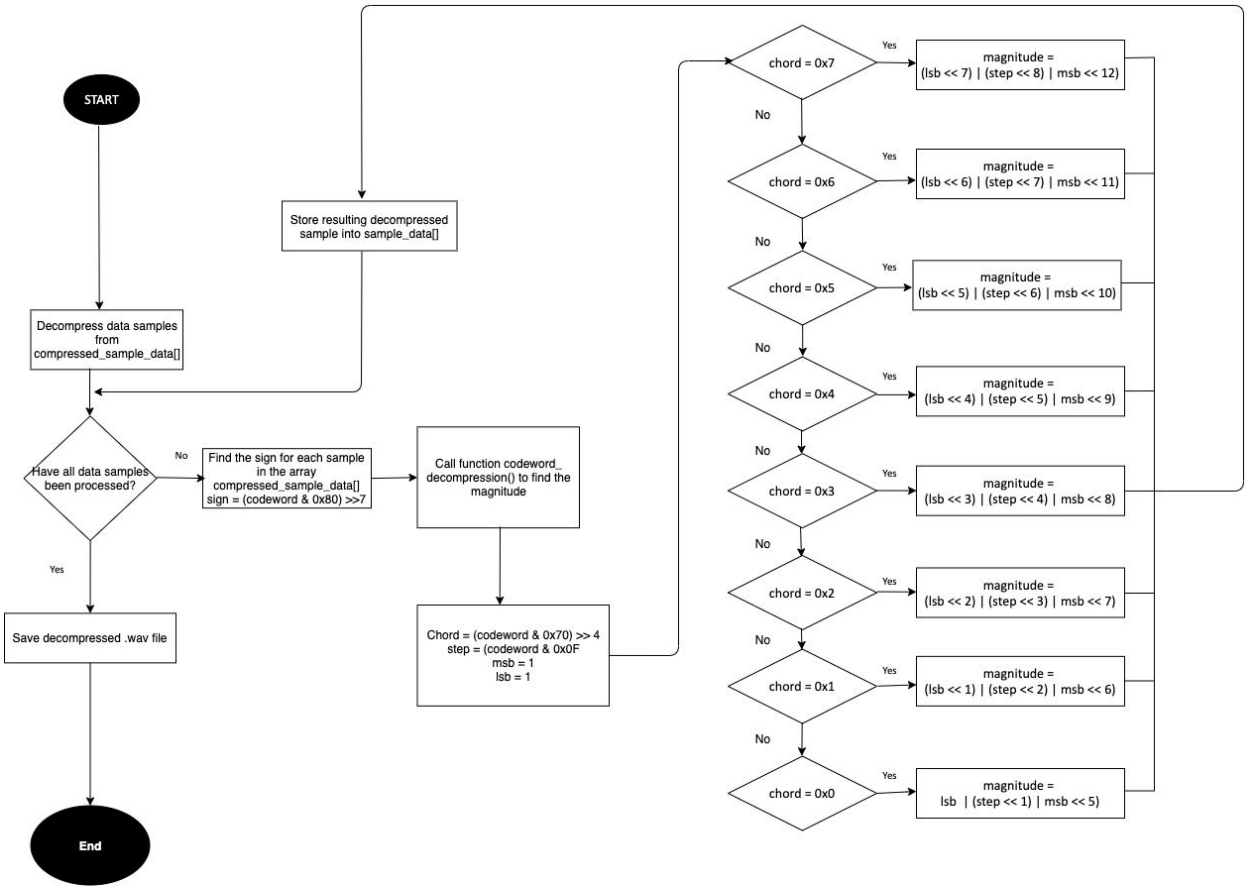

Fig 13. UML Diagram for Decompression

Appendix C - Assembly Code

The following figure displays the code snippet for important functions from our optimized Hardware assembly file - hardware_opt.s.

```
codeword_compression:
   @ Function supports interworking.
   @ args = 0, pretend = 0, frame = 0
   @ frame_needed = 1, uses_anonymous_args = 0
   @ link register save eliminated.
   str fp, [sp, #-4]!
   add fp, sp, #0
   mov r2, r0
   mov r3, #1
#APP
@ 279 "hardware_opt.c" 1
   mulaw r3, r2, r3
@ 0 "" 2
   mov r0, r3
   add sp, fp, #0
   ldmfd   sp!, {fp}
   bx  lr
   .size   codeword_compression, .-codeword_compression
   .align  2
   .global codeword_decompression
   .type   codeword_decompression, %function
codeword_decompression:
   @ Function supports interworking.
   @ args = 0, pretend = 0, frame = 32
   @ frame_needed = 1, uses_anonymous_args = 0
   @ link register save eliminated.
   str fp, [sp, #-4]!
   add fp, sp, #0
   sub sp, sp, #36
   mov r3, r0
   strb    r3, [fp, #-29]
   ldrb    r3, [fp, #-29]  @ zero_extendqisi2
   and r3, r3, #112
   mov r3, r3, asr #4
```

```
   str r3, [fp, #-24]
   ldrb    r3, [fp, #-29]  @ zero_extendqisi2
   and r3, r3, #15
   str r3, [fp, #-20]
   mov r3, #1
   str r3, [fp, #-16]
   mov r3, #1
   str r3, [fp, #-12]
   ldr r3, [fp, #-24]
   cmp r3, #7
   ldrls   pc, [pc, r3, asl #2]
   b   .L46
.L55:
   .word   .L47
   .word   .L48
   .word   .L49
   .word   .L50
   .word   .L51
   .word   .L52
   .word   .L53
   .word   .L54
.L47:
   ldr r3, [fp, #-20]
   mov r2, r3, asl #1
   ldr r3, [fp, #-12]
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #5
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L48:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #1
   ldr r3, [fp, #-20]
   mov r3, r3, asl #2
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #6
```

```
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L49:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #2
   ldr r3, [fp, #-20]
   mov r3, r3, asl #3
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #7
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L50:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #3
   ldr r3, [fp, #-20]
   mov r3, r3, asl #4
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #8
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L51:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #4
   ldr r3, [fp, #-20]
   mov r3, r3, asl #5
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #9
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L52:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #5
```

```
   ldr r3, [fp, #-20]
   mov r3, r3, asl #6
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #10
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L53:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #6
   ldr r3, [fp, #-20]
   mov r3, r3, asl #7
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #11
   orr r3, r2, r3
   str r3, [fp, #-8]
   b   .L46
.L54:
   ldr r3, [fp, #-12]
   mov r2, r3, asl #7
   ldr r3, [fp, #-20]
   mov r3, r3, asl #8
   orr r2, r2, r3
   ldr r3, [fp, #-16]
   mov r3, r3, asl #12
   orr r3, r2, r3
   str r3, [fp, #-8]
.L46:
   ldr r3, [fp, #-8]
   mov r3, r3, asl #16
   mov r3, r3, lsr #16
   mov r0, r3
   add sp, fp, #0
   ldmfd   sp!, {fp}
   bx  lr
   .size   codeword_decompression, .-codeword_decompression
```

Fig 14. Code Snippet of important functions from hardware_opt.s