Implementation and Optimization of Audio Compression & Decompression on ARM32 Processor

University of Victoria

Alex Moody (V00962257), UVic Dept. of SENG, alexmoody@uvic.ca

Daniel Alvarez (V00974015), UVic Dept. of SENG, dalvarezrivera@uvic.ca

Aug 13th, 2025

Table of Contents

Abstract

This paper presents the design, implementation, and optimization of a μ-law audio compression and decompression algorithm on an ARM32 processor. The system processes a WAV audio file and translates it into binary, applies compression and decompression techniques to the binary, and turns it back into a WAV audio file for listening. Multiple optimization strategies were explored, including software improvements and hardware-assisted operations. These optimizations were analyzed to compare and conclude which one improved the code and algorithm speed the most. Performance testing showed incremental speed gains, with software optimizations yielding greater improvements than hardware-based methods. The audio output retained most of its quality with minor static noise. The findings of this project demonstrate the use of μ-law compression in ARM32 environments and highlight the use of software and hardware optimizations.

Introduction

Specifications and Environment

For this project, we chose to use a virtual machine (VM) that virtualized an ARM32 processor environment. To do this, we used the environment provided by Professor Sima using Virtual Manager and QEMU/KTM to virtualize a Fedora 29 OS. The majority of the code was written in C, and we used the GCC compiler to compile and run the code. At the beginning, we were initially running the code in our local environments, which were not ARM32 processors, before moving all the code to the VM.

Performance Achieved

The decompressed audio recovered most of the words said in the original one, if not all of them. The voice and its tone are recognizable as well, but there is some slight fuzzy background noise, which we could not get rid of, that minorly lowers the quality and accuracy of the decompressed audio.

Contributions

- Used 16-bit integers because the WAV file was in 16-bit PCM format.
- Transformed chars to integers to be able to shift them accurately.
- Used switch cases for optimization as we thought this would be more efficient than if-else statements.
- Used the count-leading-zeroes (CLZ) assembly instruction, as it fave the exact value that we needed.

Report organization

In this report, we have broken it up into 3 main sections with more specific subsections. These sections are the introduction/background for the project, the design, implementation, and optimization of the code, and the results and conclusions we learned from the project.

Background

Audio compression and decompression are important processes in digital signal processing that enable the efficient storage and transmission of audio signals. These techniques are vital to many modern-day applications such as telephones, voice recordings, music streaming, speech recognition, and more. In embedded systems where memory, processing power, and bandwidth can be and usually are limited, the efficient implementation of audio compression and decompression is critical. We chose this project topic because we thought it would be interesting to see how it applies to these technologies we use in our everyday lives.

The main goal of this audio compression is to reduce the number of bits required to represent a signal, while maintaining and preserving an acceptable level of audio quality. In the decompression phase, the original signal gets reconstructed from its compressed form, ideally with minimal perceptual loss.

Motivation for Compression

The human auditory perception plays an important role in audio compression. Human ears perceive sound input on a logarithmic scale. As a result, compression algorithms can exploit the perceptual redundancies in audio signals. By allocating fewer bits to the insignificant components (sounds) in a signal, the algorithms reduce the total data size while still maintaining quality. To do this, the compression algorithm is implemented with a logarithm-like function, while the expansion algorithm is implemented with an exponential-like function.

Speech Signal Characteristics

Speech signals have distinct features that influence the design of compression/decompression algorithms. Speech is made up of phonemes, which are the smallest units of sound. Broadly, these can be classified into voiced phonemes, which include vowels and fricatives /v/ and /z/, and unvoiced phonemes, which include nasal consonants /m/ and /n/, fricatives /f/ and /s/, and stop consonants /p/, /t/, and /k/. The voiced phonemes' audio signal has a much larger amplitude than that of the unvoiced, roughly 10 times greater. They also have a higher probability of occurrence than the unvoiced. Therefore, the unvoiced phonemes contain much more information than the voiced ones. This imbalance in amplitude and frequency suggests that voiced signals can tolerate a much coarser quantization without significant quality degradation, which provides a good opportunity for efficient compression.

Uniform vs. Non-uniform Quantization

In traditional Uniform Pulse Code Modulation (PCM), equal-length quantization intervals are used across the full dynamic range of the signal. This approach to the problem treats all signal levels equally, which is quite inefficient for audio, where different amplitude signals require varying resolution due to the perceptual sensitivity required. To solve this issue, non-uniform quantization schemes based on logarithmic functions are used, which much better align with the variations in the human auditory system sensitivity. These schemes compress the dynamic range of the input signal, reducing

the number of bits required to represent it. This technique is the one we will be employing in this project.

μ-Law and A-Law for Logarithmic Compression

There are two widely adopted and accepted logarithmic compressions for audio signals, and they are the μ-law and A-law. The μ-law is primarily used in North America and Japan, whereas the A-law is used in Europe and other regions. For this project, we will be using μ-law, as it is more applicable to North America, where we are writing this report.

These laws allow large signals to be reduced to and represented with fewer bits while smaller signals still retain more detail. As a result, higher perceptual quality is achieved at lower bitrates. The logarithmic PCM allows 8 bits per sample to represent the same values that would be represented in 14 bits in a sample using uniform PCM. This logarithmic PCM then produces a compression ratio of 1.75 : 1.]

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

μ-Law quantizer (PCM-to-μ-law). Where the argument 0 ≤ |x| ≤ 1 and μ is a parameter which ranges from 0 (no compression) to 255 (maximum compression) [1].

$$y = \begin{cases} \operatorname{sgn}(x)\dfrac{A|x|}{1+\ln A} & \text{for } 0 \le |x| \le \dfrac{1}{A} \\ \operatorname{sgn}(x)\dfrac{1+\ln(A|x|)}{1+\ln A} & \text{for } \dfrac{1}{A} \le |x| \le 1 \end{cases}$$

A-Law quantizer (PCM-to-A-Law). Where A = 87.6 and x is the normalized integer to be compressed [1].

Algorithm Design

Software Design

Software Architecture and Algorithm Flow

The compression/decompression system is built around the μ-law algorithm; a non-uniform quantization technique. The implementation of the algorithm uses 16-bit PCM audio data and produces 8-bit μ-law codewords to achieve the desired data size reduction. They are then returned back to 16-bit size and into a WAV file for listening. The design consists of three main stages: preprocessing, compression, and decompression.

WAV File Processing (Preprocessing)

The WAV file was processed using a python script. The script extracts the 16-bit PCM binary from the WAV file, normalizes it to store the sample in binary, and then writes it into a text file used as input for the μ Law Audio Compression Algorithm.

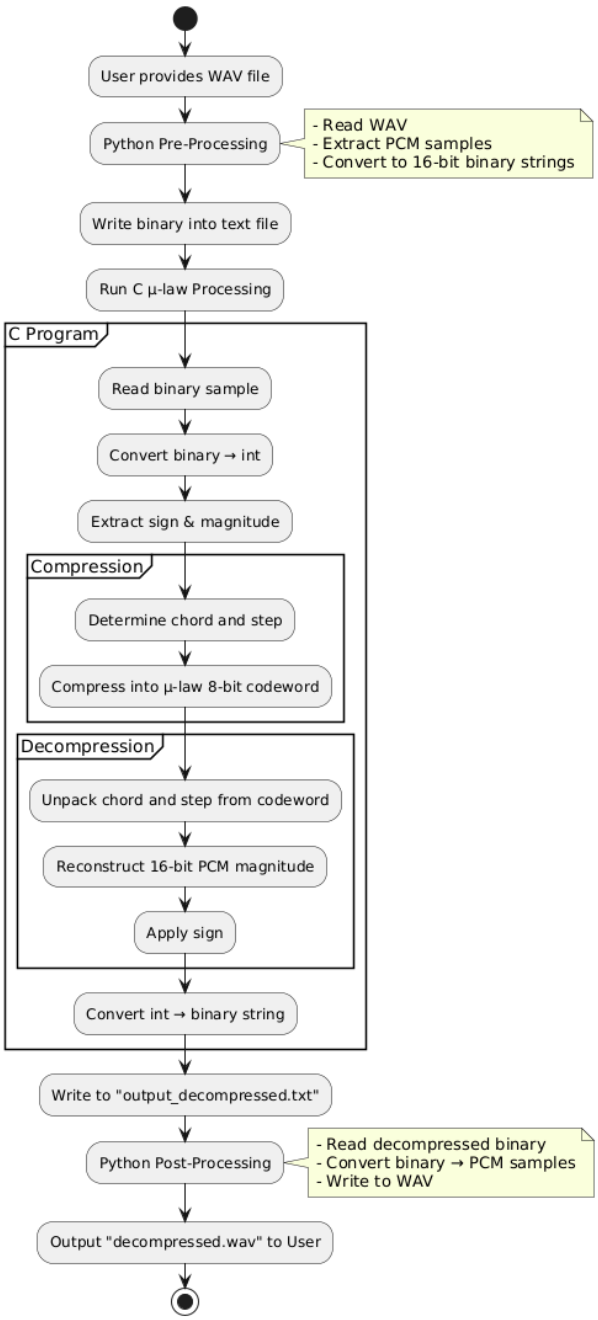
μ Law Audio Compression Algorithm

The compression algorithm converts each 16-bit PCM sample into an 8-bit μ-law codeword. First, the sign and magnitude are extracted. This is done by isolating the sign bit and obtaining the magnitude as the absolute value of the sample. Next, the chord and step are calculated. The magnitude is classified into one of eight chords based on the position of the most significant 1-bit. This is done with if-else statements initially and later with switch cases. The step is then calculated from the lower bits following after the chord. We initially used masking and shifting for this operation before moving to CLZ later. The sign, chord, and step are then put into an 8-bit codeword using bitwise operations.

μ Law Audio Decompression Algorithm

The decompression algorithm reconstructs a 16-bit PCM value from the 8-bit codeword obtained during compression. The algorithm extracts the sign, chord and step values from the codeword. Then, a base pattern of ones and zeroes is assigned based on the chord value using if-else statements and later switch cases. The step value is shifted into place afterwards. Lastly, the sign is applied to the sample and is negated if the sign bit has a value of zero.

The algorithms are executed sequentially in a loop that processes each sample. The algorithm is shown below in a UML activity diagram that outlines the flow and sequence of our software design (the loop is left out for clarity but would essentially do everything in the “C Program” box as well as the writing of output to the text file.).



C code

Prior to the compression and decompression algorithms being called, we obtained the sign and magnitude values. That code is shown below:

```
int signum (int sample){
    if(sample < 0){
        return(0) ; /* sign is '0' for negative samples */
    }else {
        return(1) ; /* sign is '1' for positive samples */
    }
}

int get_magnitude (int sample){
    if(sample < 0) {
        sample = -sample;
    }
    return(sample);
}
```

Here is our initial C code for the compression and decompression algorithms. We used the example code given to us in the slides as a base for the initial compression, except for the calculation of codeword_tmp, which was altered to change the sign of the number but is still very similar [1].

```
unsigned char codeword_compression(unsigned int sample_magnitude, int sign) {
    int chord = 0;
    int step = 0;
    unsigned char 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 if (sample_magnitude & (1 << 5)) {
        chord = 0x0;
        step = (sample_magnitude >> 1) & 0xF;
    } else {
        chord = 0x0;
        step = 0x0;
    }

    codeword_tmp = ((abs(sign - 1)) << 7) | (chord << 4) | step;
    return codeword_tmp;
}
```

For the decompression algorithm, we used bit shifting and masking to get the chord value, and using that value, we decompressed each case. We also used a number containing the zeros and ones according to each chord, based on the values represented in the lookup table in the slides [1]. For the sign bit, after several attempts to switch signs as needed due to the codeword inversion during compression, we found this way of inverting it to be the most effective in terms of our output.

```
int16_t codeword_decompression(unsigned char compressed) {
    int step = compressed & 0x0F;
    int codeword_tmp = 1;
    int chord = (compressed & 0x70) >> 4;
    int sign = (compressed & 0x80) >> 7;
    int zeros_ones;

    if (chord == 7){
        zeros_ones = 0x80;
```

```
        codeword_tmp = (1 << 12) | (step << 8) | zeros_ones;
    } else if (chord == 6){
        zeros_ones = 0x840; // Rightmost zeros
        codeword_tmp = (step << 7) | zeros_ones;
    } else if (chord == 5){
        zeros_ones = 0x420; // Rightmost zeros
        codeword_tmp = (step << 6) | zeros_ones;
    } else if (chord == 4){
        zeros_ones = 0x210; // Rightmost zeros
        codeword_tmp = (step << 5) | zeros_ones;
    } else if (chord == 3){
        zeros_ones = 0x108; // Rightmost zeros
        codeword_tmp = (step << 4) | zeros_ones;
    } else if (chord == 2){
        zeros_ones = 0x84; // Rightmost zeros
        codeword_tmp = (step << 3) | zeros_ones;
    } else if (chord == 1){
        zeros_ones = 0x102; // Rightmost zeros
        codeword_tmp = (step << 2) | zeros_ones;
    } else if (chord == 0){
        zeros_ones = 0x21; // Rightmost zeros
        codeword_tmp = (step << 1) | zeros_ones;
    }

    int16_t sample;
    if (sign == 0) {
        sample = -(int16_t)codeword_tmp;
    } else {
        sample = (int16_t)codeword_tmp;
    }
    return sample;
}
```

Optimization

Provided here is the decompression algorithm that uses switch cases for optimization instead of if-else statements.

For this, we followed the same process as with the if-else, but with switch statements. We chose to implement switches because they were mentioned in the slides as an

optimization idea. This is because the code can jump to the correct case instead of checking each if-else statement.

```
// μ-law decompression
int16_t codeword_decompression(unsigned char compressed) {
    int step = compressed & 0x0F;
    int codeword_tmp = 1;
    int chord = (compressed & 0x70) >> 4;
    int sign = (compressed & 0x80) >> 7;
    int zeros_ones;

    switch (chord){
        case 7:
            zeros_ones = 0x80;
            codeword_tmp = (1 << 12) | (step << 8) | zeros_ones; break;
        case 6:
            zeros_ones = 0x840; // Rightmost zeros
            codeword_tmp = (step << 7) | zeros_ones; break;
        case 5:
            zeros_ones = 0x420; // Rightmost zeros
            codeword_tmp = (step << 6) | zeros_ones; break;
        case 4:
            zeros_ones = 0x210; // Rightmost zeros
            codeword_tmp = (step << 5) | zeros_ones; break;
        case 3:
            zeros_ones = 0x108; // Rightmost zeros
            codeword_tmp = (step << 4) | zeros_ones; break;
        case 2:
            zeros_ones = 0x84; // Rightmost zeros
            codeword_tmp = (step << 3) | zeros_ones; break;
        case 1:
            zeros_ones = 0x102; // Rightmost zeros
            codeword_tmp = (step << 2) | zeros_ones; break;
        case 0:
            zeros_ones = 0x21; // Rightmost zeros
            codeword_tmp = (step << 1) | zeros_ones; break;
    }
    int16_t sample;
    if (sign == 0) {
        sample = -(int16_t)codeword_tmp;
    } else {
        sample = (int16_t)codeword_tmp;
```

```
    }
    return sample;
}
```

Compile Code and Assembly Optimization

Here is our compilation code and output:

```
[root@localhost SENG_440_Project]# gcc compressV2.c -o compressV2
[root@localhost SENG_440_Project]# ./compressV2
Opened input file
Opened putput file
Before while loop.
Complete
Audio Compression and decompression Time:1.882118  Seconds
µ-law compression & decompression done.
```

Below is the compression algorithm using the CLZ assembly instruction for optimization. The CLZ (Count Leading Zeros) instruction in assembly counts the number of leading zeros in a binary representation of a number. We picked this method as it intuitively gives the values needed for this process, just as it is shown in the lookup table. Then, it's the same process as with the previous software solution.

```
int codeword_compression_clz(unsigned int sample_magnitude, int sign) {
    int chord, step;
    int codeword_tmp;
    int zeros_at_left_side = __builtin_clz(sample_magnitude << 19);

    if(zeros_at_left_side == 0){
        chord = 0x7;
        step = (sample_magnitude >> 8) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    if(zeros_at_left_side == 1){
        chord = 0x6;
        step = (sample_magnitude >> 7) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
```

```
    }
    if(zeros_at_left_side == 2){
        chord = 0x5;
        step = (sample_magnitude >> 6) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    if(zeros_at_left_side == 3){
        chord = 0x4;
        step = (sample_magnitude >> 5) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    if(zeros_at_left_side == 4){
        chord = 0x3;
        step = (sample_magnitude >> 4) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    if (zeros_at_left_side == 5){
        chord = 0x2;
        step = (sample_magnitude >> 3) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    if(zeros_at_left_side == 6){
        chord = 0x1;
        step = (sample_magnitude >> 2) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    if (zeros_at_left_side == 7){
        chord = 0x0;
        step = (sample_magnitude >> 1) & 0xF;
        codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
    }
    return (int) sample_magnitude;
}
```

Below is the compression algorithm using the CLZ assembly instruction and switch cases for optimization as well.

In this case, we did the same as the previous software solutions, but with switch statements, because as explained before, they are faster than if-else statements.

```
int codeword_compression_clz_switch(unsigned int sample_magnitude, int sign) {
    int chord, step;
    int codeword_tmp;
    int zeros_at_left_side = __builtin_clz(sample_magnitude << 19);

    switch (zeros_at_left_side){
        case 0:
            chord = 0x7;
            step = (sample_magnitude >> 8) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
            return (codeword_tmp);
        case 1:
            chord = 0x6;
            step = (sample_magnitude >> 7) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
            return (codeword_tmp);
        case 2:
            chord = 0x5;
            step = (sample_magnitude >> 6) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
            return (codeword_tmp);
        case 3:
            chord = 0x4;
            step = (sample_magnitude >> 5) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
            return (codeword_tmp);
        case 4:
            chord = 0x3;
            step = (sample_magnitude >> 4) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
        return (codeword_tmp);
        case 5:
            chord = 0x2;
            step = (sample_magnitude >> 3) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
```

```
            return (codeword_tmp);
        case 6:
            chord = 0x1;
            step = (sample_magnitude >> 2) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
            return (codeword_tmp);
        case 7:
            chord = 0x0;
            step = (sample_magnitude >> 1) & 0xF;
            codeword_tmp = ((abs(sign-1)) << 7) | (chord << 4) | step;
            return (codeword_tmp);
        default:
            return (int) sample_magnitude;
    }
}
```

Results

The resulting audio has recovered all of the words said in the original recording. The voice and its tone are recognizable as well, but there is some minor static noise overall. So, the resulting audio is understandable and audible, and quality and accuracy of the decompressed audio are just slightly lower than desired.

Here is a list of the times recorded in the virtual machine environment when running the code. The times for each were recorded 10 times, and then an average time was taken from those.

- No optimization: 2.055 Seconds
- With CLZ in compression: 2.00655 Seconds (-2.36%)
- With CLZ and switch in compression: 1.935 Seconds (-3.55%)
- With the above and switch statements in decompression: 1.856 Seconds (-4.07%)

As you can see, the time did go down slightly with each optimization implemented, and utilizing Hardware did not have as big an impact on the time as the Software optimization (switch statements). You can listen to the audio sample input and output in the presentation for this project as well.

Improvements

The main improvement we could make on this project is lowering the static noise. But we do not know if it's possible because it's minimal, and it might be occurring because of the information lost due to the compression/decompression process, and there may be nothing we can do about it.

Unfortunately, we did not manage to write our own assembly instruction for this Project, due to some confusion with the CLZ instruction; we were not sure if it was a requirement. Based on our progress meetings during the term we were under the impression that using the CLZ instruction for optimization was sufficient enough. We also struggled to get the original implementation working (without optimizations) as soon as we would have liked in order to have more time for work on optimizing the algorithms.

Conclusions

Finally, we can conclude that our implementation of the μ-Law is effective for decompression and compression, although it may have some slight static noise and room for improvement. However, the majority of the message was kept at high quality, and is very understandable. The software optimizations turned out to be more effective than the hardware ones, as we could see in the Results section, switch statements lowered the time more than the use of the CLZ instruction.

References

[1] M. Sima, “SENG440 Embedded Systems – Lesson 104: Audio Compression –.”

Appendix

A.1 - UML

A.1.1 - Activity Diagram

```
@startuml
'title μ-law Audio Compression/Decompression\n(Activity Diagram)\n

start

:User provides WAV file;

:Python Pre-Processing;
note right
  - Read WAV
  - Extract PCM samples
  - Convert to 16-bit binary strings
end note

:Write binary into text file;

:Run C μ-law Processing;

partition "C Program" {
    :Read binary sample;
    :Convert binary → int;
```

```
    :Extract sign & magnitude;

    partition "Compression" {
        :Determine chord and step;
        :Compress into μ-law 8-bit codeword;
    }

    partition "Decompression" {
        :Unpack chord and step from codeword;
        :Reconstruct 16-bit PCM magnitude;
        :Apply sign;
    }

    :Convert int → binary string;
}

:Write to "output_decompressed.txt";

:Python Post-Processing;
note right
  - Read decompressed binary
  - Convert binary → PCM samples
  - Write to WAV
end note

:Output "decompressed.wav" to User;

stop
@enduml
```

A.2 - Revisions

These are revisions made based on the professors requests and questions during our project presentation on August 14th, 2025.You may also find these changes appended to the Project Report as well.

Revision 1 - Switch Case Assembly

During the presentation, Dr. Sima asked how the switch cases we implemented looked in assembly and whether the way they were implemented was the best way to optimize the code.
Here is the assembly code for the switch cases in the decompression function:

```
codeword_decompression_switch:
    pushq    %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe    %rbp, 0
    subq    $32, %rsp
    .seh_stackalloc    32
    .seh_endprologue
    movl    %ecx, %eax
    movb    %al, 16(%rbp)
    movzbl    16(%rbp), %eax
    andl    $15, %eax
    movl    %eax, -12(%rbp)
    movl    $1, -4(%rbp)
    movzbl    16(%rbp), %eax
    sarl    $4, %eax
    andl    $7, %eax
    movl    %eax, -16(%rbp)
    movzbl    16(%rbp), %eax
```

```
    shrb    $7, %al
    movzbl    %al, %eax
    movl    %eax, -20(%rbp)
    cmpl    $7, -16(%rbp)
    je    .L62
    cmpl    $7, -16(%rbp)
    jg    .L63
    cmpl    $6, -16(%rbp)
    je    .L64
    cmpl    $6, -16(%rbp)
    jg    .L63
    cmpl    $5, -16(%rbp)
    je    .L65
    cmpl    $5, -16(%rbp)
    jg    .L63
    cmpl    $4, -16(%rbp)
    je    .L66
    cmpl    $4, -16(%rbp)
    jg    .L63
    cmpl    $3, -16(%rbp)
    je    .L67
    cmpl    $3, -16(%rbp)
    jg    .L63
    cmpl    $2, -16(%rbp)
    je    .L68
    cmpl    $2, -16(%rbp)
    jg    .L63
    cmpl    $0, -16(%rbp)
    je    .L69
    cmpl    $1, -16(%rbp)
    je    .L70
```

```
  jmp   .L63
```

The section highlighted in yellow is the switch cases for our C code. As you can see it is not using a switch jump table and is instead branching. This is not ideal and not as fast as the jump table. It is realistically not that much faster than using if-else statements and therefore is not the most optimal way to optimize the code. Ideally, we should have specified in assembly code embedded into our C code to use a jump table instead. This would have been better for the optimizations and likely would've led to better results.

Revision 2 - CLZ Instruction Vector Operations

Here is the use of the Vector CLZ method. We would implement it in this manner (code attached below), where the sample magnitude contains 4 integers of 16 bits each, and then, with a loop, each magnitude is compressed and stored in another array.

```
// // μ-law compression with CLZ
int codeword_compression_clz(uint16x4_t sample_magnitude, int *sign, int size) {
   int chord, step;
   int codeword_tmp[size];
   // Count leading zeros per element
   uint16x4_t zeros_at_left_side = vclz_u16(sample_magnitude);

   uint16_t zeros_array[size];
   vst1_u16(zeros_array, zeros_at_left_side);
   for(int i = 0; i < size; i++){
       if(zeros_array[i] == 0){
           chord = 0x7;
           step = (sample_magnitude >> 8) & 0xF;
           codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
           continue;
       }
       if(zeros_array[i] == 1){
           chord = 0x6;
           step = (sample_magnitude >> 7) & 0xF;
           codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
           continue;

       }
       if(zeros_array[i] == 2){
```

```
            chord = 0x5;
            step = (sample_magnitude >> 6) & 0xF;
            codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
            continue;
        }
        if(zeros_array[i] == 3){
            chord = 0x4;
            step = (sample_magnitude >> 5) & 0xF;
            codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
            continue;
        }
        if(zeros_array[i] == 4){
            chord = 0x3;
            step = (sample_magnitude >> 4) & 0xF;
            codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
            continue;
        }
        if (zeros_array[i] == 5){
            chord = 0x2;
            step = (sample_magnitude >> 3) & 0xF;
            codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
            continue;
        }
        if (zeros_array[i] == 6){
            chord = 0x1;
            step = (sample_magnitude >> 2) & 0xF;
            codeword_tmp[i] = ((abs(sign[i]-1)) << 7) | (chord << 4) | step;
            continue;
        }
        if (zeros_array[i] == 7){
            chord = 0x0;
            step = (sample_magnitude >> 1) & 0xF;
            codeword_tmp[i] = ((abs(sig[i]n-1)) << 7) | (chord << 4) | step;
        }
    }
    return (codeword_tmp);
}
```

Revision 3 - Equivalence Test Redundancy

In one of our functions we have the following code:

```
    int16_t sample;
    if (sign == 0) {
        sample = -(int16_t)codeword_tmp;
    } else {
        sample = (int16_t)codeword_tmp;
    }
    return sample;
```

This code is not optimized, or as safe as it could be. The sign == 0 is unnecessary in this case. Better code would be as follows:

```
    int16_t sample;
    if (sign) {
        sample = -(int16_t)codeword_tmp;
    } else {
        sample = (int16_t)codeword_tmp;
    }
    return sample;
```

Here we have changed “if (sign == 0)” to “if (sign)” which does the same thing but is better coding practice in C.