University Of Victoria
Department of Engineering

SENG 440 Project Report
Summer 2015

24-bit to 16-bit Audio Compression

Trison Nguyen
V00178742
trison@uvic.ca

James Robertson
V0075922
jamestom@uvic.ca

July 31, 2015

Table Of Contents

1 Introduction

Audio compression is widely used to reduce storage and transmission of audio data. This is commonly done by using either the μ-law or A-law quantization methods. Both methods are highly desirable in audio compression because they approximate a scaled logarithm function. The human auditory system has a dynamic response that is shaped similarly to a logarithm. Using a compression method where quantization steps remain constant then results in larger relative quantization error for smaller level signals[1]. Using a logarithmic compression method allows for an efficient form of compression where a higher compression ratio is applied to dynamic ranges where changes are less sensitive to the human ear. Ranges that are more sensitive to human hearing then results in having less compression applied.

This project utilizes a piecewise logarithmic approximation to implement the compression method. Our algorithm was required to compress 24-bit signals to 16-bit signals. After initial implementation of the algorithm in C, optimizations to increase code efficiency were made. These optimizations were measured to see how much time improvements could be made. Several optimizations were implemented with the original C program, as well as in the assembly domain with the resulting generated assembly code. A survey of the assembly code revealed the effectiveness and importance of optimizations.

This report details the μ-law quantization method, the piecewise approximation and its implementation, methods of code optimization and the outcome of our work.

2 Theoretical Background

The human auditory system behaves similarly to a logarithmic function; low level signals are perceived to be more susceptible to changes in resolution as opposed to high level signals[1]. Compression methods that utilize uniform compression are then not feasible due to their poor functionality at low signal levels when also considering compression efficiency. Audio compression must then utilize logarithmic functions according to human auditory perception. Two common methods for audio compression are μ-law and A-law. Both methods are primarily used to reduce bandwidth in transmission of telephony signals. A-law is more standardly employed in Europe, while μ-law is the standard compression method in North America and Japan[2]. The formulas for these algorithms can be seen below:

$$F(x) = \operatorname{sgn}(x) \begin{cases} \frac{A|x|}{1+\ln(A)}, & |x| < \frac{1}{A} \\ \frac{1+\ln(A|x|)}{1+\ln(A)}, & \frac{1}{A} \leq |x| \leq 1, \end{cases}$$

Figure 1. A-Law Compression

$$F(x) = \operatorname{sgn}(x) \frac{\ln(1+\mu|x|)}{\ln(1+\mu)} \qquad -1 \leq x \leq 1$$

Figure 2. μ-Law Compression

Using the μ-law algorithm results in a decreased dynamic range of the original signal, but has a better signal-to-noise ratio when compared to an algorithm that compresses linearly[3].

This project implements a piecewise approximation of μ-law compression. The interval steps were divided into powers of 2 for a total 16 step(17 including error case) levels of compression according to our requirement of compression from 24 bits into 16 bits.

3 Design Process

3.1 Implementation in C

Our initial design step was to implement the μ-law algorithm in C. To represent a simplified audio stream, an array with 5 samples was generated. The samples were distributed across a space that encompasses the full range of compression, ie. 0 to 2^{24}.

```
unsigned int samples[] = {25, 300, 6500, 4000000, 16777215};
```

Figure 3. Initialize array of samples

Each sample was then bitwise shifted by 8 in order to retrieve the most significant 16 bits. The shifted samples were then sent to our piecewise logarithmic approximation function for compression. After compression, the result is combined with the remaining values from our sample, completing the 24-bit to 16-bit compression.

```
unsigned long func(unsigned long val){

    unsigned long compressed = pwlog2(val >> 8);
    unsigned long rem = val >> ((compressed >> 12) +2);
    unsigned long final = compressed | rem;
    return final;
}
```

Figure 4. Combination with compressed result

The piecewise approximation is done in our function `pwlog2()`. The implementation was done with guidance from the 8-bit logarithmic compression example as demonstrated in Slide Lesson 101[1]. Here, we adjusted the algorithm to incorporate 24-bit to 16-bit logarithmic compression, eg. an input value with max value 16,777,215, to max value 65,536.

```
static inline unsigned long pwlog2(unsigned long x){
    if( x >= 32768){/* range 2^25 to 2^16 */
        return( (61440) + ((x-32768)>>3));
    }
    if( x >= 16384){
        return( (57344) + ((x-16384)>>2));
    }
    if( x >= 8192){
        return( (53248) + ((x-8192)>>1));
    }
    if( x >= 4096){
        return( (49152) + ((x-4096)));
    }
    if( x >= 2048){
        return( (45056) + ((x-2048)<<1) );
    }
    if( x >= 1024){
        return( (40960) + ((x-1024)<<2));
    }
    if( x >= 512){
        return( (36864) + ((x-512)<<3));
    }
    if( x >= 256){
        return( (32768) + ((x-256)<<4));
    }
    if( x >= 128){
        return( (28672) + ((x-128)<<5));
    }
    if( x >= 64){
        return( (24576) + ((x-64)<<6));
    }
    if( x >= 32){
        return( (20480) + ((x-32)<<7));
    }
    if( x >= 16){
        return((16384) + ((x-16)<<8));
    }
    if( x >= 8){
        return((12288) + ((x-8)<<9));
    }
    if( x >= 4){
        return((8192) + ((x-4)<<10));
    }
    if( x >= 2){
        return((4096) + ((x-2)<<11));
    }
    if( x >= 1){
        return((x-1) << 12);
    }
    if( x == 0){
        return( 0); /* error */
    }
}
```

Figure 5. Piecewise log approximation

3.2 C Code Optimization

Our functioning C code was further optimized in order to improve performance. Several optimization methods were used: reversing the order of intervals, use a lookup table, return constant values (eg. no calculations), and combinations of the above. Our initial implementation can be found in main.c, and each optimization can be found in the main_opt*.c files.

3.2.1 Reverse order of intervals

This optimization of the C code included reversing the order of `if` statements in our piecewise approximation. By now checking the largest range of intervals first, we reduce the number of iterations through the `if` statements. The first interval then encompasses the largest range of values, resulting in half of a uniformly distributed set of possible values hitting and returning from the first `if` statement. The code snippet below illustrates our implementation:

```
unsigned long pwlog2(unsigned long x){
    if( x >= 32768){/* range 2^16 to 2^24 */
        return( (15<<12) + ((x-32768)>>3));
    }
    if( x >= 16384){
        return( (14<<12) + ((x-16384)>>2));
    }
    if( x >= 8192){
        return( (13<<12) + ((x-8192)>>1));
    }
    if( x >= 4096){
        return( (12<<12) + ((x-4096)));
    }
    if( x >= 2048){
        return( (11<<12) + ((x-2048)<<1) );
    }
    if( x >= 1024){
        return( (10<<12) + ((x-1024)<<2));
    }
    if( x >= 512){
        return( (9<<12) + ((x-512)<<3));
    }
    if( x >= 256){
        return( (8<<12) + ((x-256)<<4));
    }
    if( x >= 128){
        return( (7<<12) + ((x-128)<<5));
    }
    if( x >= 64){
        return( (6<<12) + ((x-64)<<6));
    }
    if( x >= 32){
        return( (5<<12) + ((x-32)<<7));
    }
    if( x >= 16){
        return((4<<12) + ((x-16)<<8));
    }
    if( x >= 8){
        return((3<<12) + ((x-8)<<9));
    }
    if( x >= 4){
        return((2<<12) + ((x-4)<<10));
    }
    if( x >= 2){
        return((1<<12) + ((x-2)<<11));
    }
    if( x >= 1){
        return((x-1) << 12);
    }
    if( x == 0){
        return( 0); /* error */
    }
}
```

Figure 6. Reversed intervals

3.2.2 Lookup Table

This optimization method used a lookup table in order to return a compressed value. The shifting, addition and subtraction operations were then replaced with a simple lookup to an array of constant values. The lookup array was composed of approximated logarithmic values for inputs ranging from 0 to 2^{24}.

```
unsigned long lookup[] = {
    16,
    15,
    14,
    13,
    12,
    11,
    10,
    9,
    8,
    7,
    6,
    5,
    4,
    3,
    2,
    1,
    0
};

/* pwlog2 = piecewise log2 */
/* 2nd optimization - use lookup table */
unsigned long pwlog2(unsigned long x){
    if( x >= 32768){/* range 2^16 to 2^24 */
        return lookup[0];
    }
    if( x >= 16384){
        return lookup[1];
    }
    if( x >= 8192){
        return lookup[2];
```

Figure 7. Lookup Table

3.2.3 Return Constants

This optimization method essentially further optimizes the lookup table method; now return values are placed directly in the `pwlog2()` function, removing the need for an additional lookup call for each `if` case. The `pwlog2()` function then returns constants and no further calculations and operations are needed for computation.

```
if( x >= 32768){/* range 2^16 to 2^24 */
    return 16;
}
if( x >= 16384){
    return 15;
}
if( x >= 8192){
    return 14;
}
if( x >= 4096){
    return 13;
}
if( x >= 2048){
    return 12;
}
if( x >= 1024){
    return 11;
}
if( x >= 512){
    return 10;
}
if( x >= 256){
    return 9;
}
if( x >= 128){
    return 8;
}
```

Figure 8. Constant return values

3.2.4 Function Inlining and Operator Strength Reduction

This optimization method combines reverse order of intervals with function inlining and operator strength reduction. Function inlining was implemented by including `static inline` before the `pwlog2()` function declaration. Operator strength reduction was implemented by replacing the bit shifting required to determine each interval's scale value with just a constant value. The initial implementation is then optimized with few operations and faster lookups. This final optimized version is what we will mainly compare with the original implementation.

```
static inline unsigned long pwlog2(unsigned long x){
    if( x >= 32768){/* range 2^16 to 2^24 */
        return( (61440) + ((x-32768)>>3));
    }
    if( x >= 16384){
        return( (57344) + ((x-16384)>>2));
    }
    if( x >= 8192){
        return( (53248) + ((x-8192)>>1));
    }
    if( x >= 4096){
        return( (49152) + ((x-4096)));
    }
    if( x >= 2048){
        return( (45056) + ((x-2048)<<1) );
    }
    if( x >= 1024){
        return( (40960) + ((x-1024)<<2));
    }
    if( x >= 512){
        return( (36864) + ((x-512)<<3));
    }
    if( x >= 256){
        return( (32768) + ((x-256)<<4));
    }
    if( x >= 128){
        return( (28672) + ((x-128)<<5));
```

Figure 9. Function Inlining w/ Operator Strength Reduction

3.3 VHDL

The entity audioc in VHDL code is made up of two ports our input vector `val`, which is 24 bits and output vector `final` of 16bits. The clock was omitted in this case because we have could not compile the results.

```
entity audioc is
    Port    val :       in STD_LOGIC_VECTOR(23 downto 0);
           final :     out STD_LOGIC_VECTOR (15 downto 0);
end audioc;
```

Figure 10. VHDL port definitions

3.3.1 Implementation

The implementation consisted of making a process for `powerlog2()` which had the input of the initial 24-bit vector shifted bits to the right. Binary bits and an and operator were used to

compare the input vector. and a binary number was also used as part of the output. The VHDL code can be seen below.

```
powerlog2 process(val srl 8)
begin:
    if    (x and "1000000000000000"/=0) then
        result <=( ("1111000000000000") + ((x-32768) srl 3));
    elsif (x and "1100000000000000"/=0) then
        result <=( ("1110000000000000") + ((x-16384) srl 2));
    elsif (x and "1010000000000000"/=0) then
        result <=( ("1101000000000000") + ((x-8192) srl 1));
    elsif (x and "1001000000000000"/=0) then
        result <=( ("1100000000000000") + ((x-4096)));
    elsif(x and "1000100000000000"/=0) then
        result <=( ("1011000000000000") + ((x-2048) sll 1) );
    elsif (x and "1000010000000000"/=0) then
        result <=( ("1010000000000000") + ((x-1024) sll 2));
    elsif (x and "1000001000000000"/=0) then
        result <=( ("1001000000000000") + ((x-512) sll 3));
    elsif (x and "1000000100000000"/=0) then
        result <=( ("1000000000000000") + ((x-256) sll 4));
    elsif (x and "1000000010000000"/=0) then
        result <=( ("0111000000000000") + ((x-128) sll 5));
    elsif (x and "1000000001000000"/=0) then
        result <=( ("0110000000000000") + ((x-64) sll 6));
    elsif (x and "1000000000100000"/=0) then
        result <=( ("0101000000000000") + ((x-32) sll 7));
    elsif (x and "1000000000100000"/=0) then
        result <=(("0100000000000000") + ((x-16) sll 8));
    elsif (x and "1000000000010000"/=0) then
        result <=(("0011000000000000") + ((x-8) sll 9));
    elsif (x and "1000000000001000"/=0) then
        result <=(("0010000000000000") + ((x-4) sll 10));
    elsif (x and "1000000000000100"/=0) then
        result <=(("0001000000000000") + ((x-2) sll 11));
    elsif (x and "1000000000000010"/=0) then
        result =((x-1) sll 12);
    elsif x = 0 then
        result <= "0000000000000000";
    end if;
end process;
```

Figure 11. VHDL code snippet

3.4 New assembly instruction

If we could implement the hardware designed in the previous section we would use the code below . This new instruction `powerlog2()` would have an input of `val` and output `compressed`.

```
unsigned int compressed, val;
_asm_("powerlog2 %0, %1" : "=r" (compressed) : "r"(val));
```

Figure 12 . New Assembly Instruction

3.5 UML Diagram

The UML diagram was helpful in the design process, it gave more of a visual aspect and helped with both troubleshooting and completing the code. The UML Diagram is attached in the appendix part of it are shown below with a brief explanation.

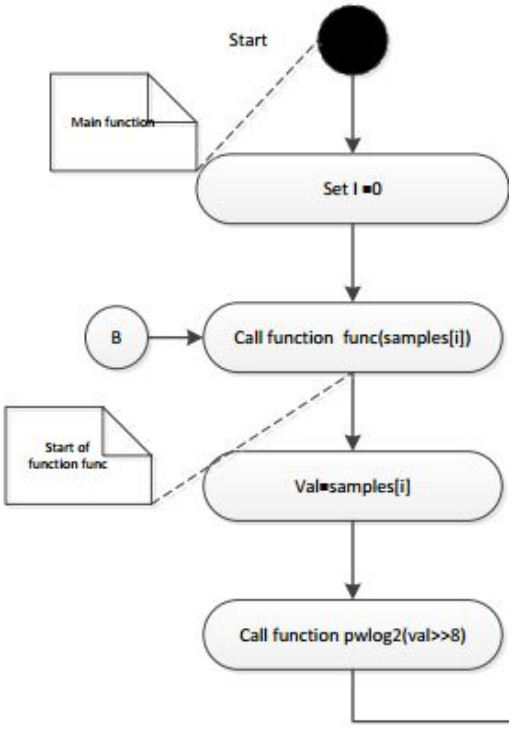

Figure 13. UML Section 1

In the figure show above we have the start of the audio compression with the initial value of `i` set to 0.The function func was called with the input `samples[i]`, which expresses the audio to be compressed. The variable `val` is then set to the input audio and shifted right by 8 bits. This variable is now the input for the function `pwlog2()`. Shown in the figure below is the implementation of the function `pwlog2()`.

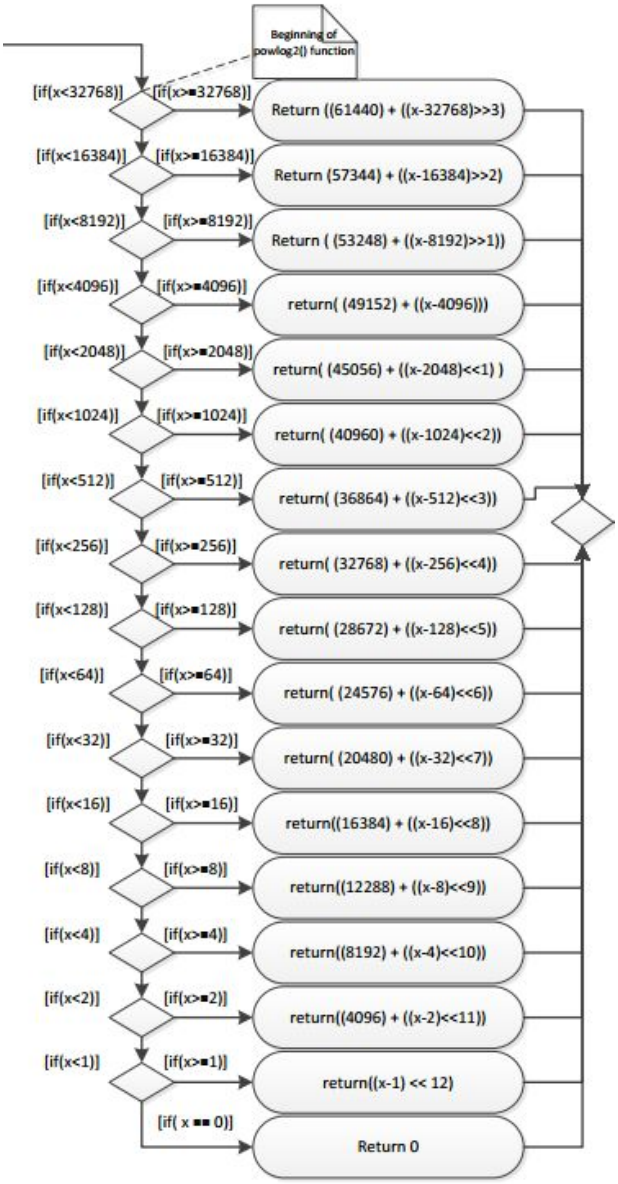

Figure: 14. UML Section 2

After a value is returned from the `pwlog2()` we re-enter the function func and the returned value is stored in the variable `compressed`. Then the remainder is found and stored in `rem` and is logically OR'd with the value `compressed` to find a final value which is then stored in the array called `results[i]`. After this step `i` is incremented and the `for` loop checks whether is is larger than the sample size. If it is larger then the program terminated if not the next sample value is obtained and the process repeats from B.

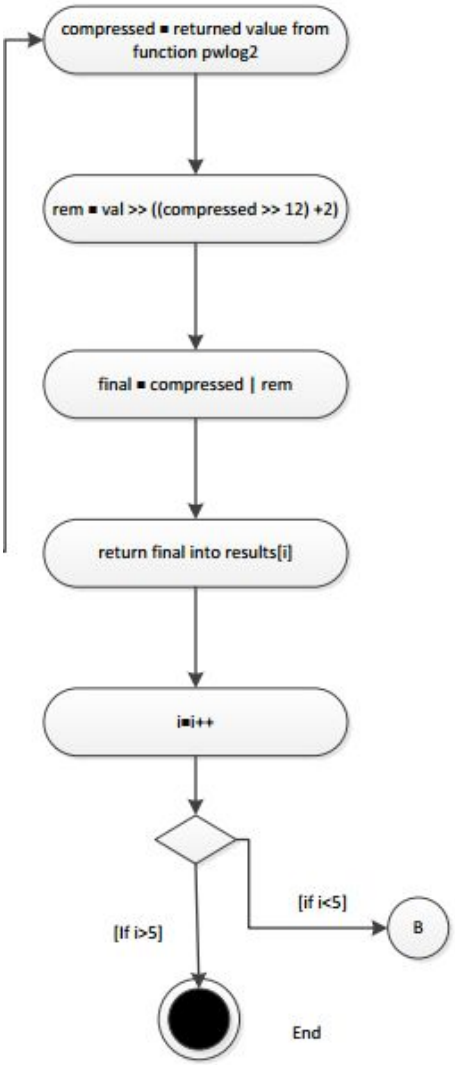

Figure 15. UML Section 3

4 Performance and Cost Evaluation

4.1 Software Evaluation:

We determined the performance improvement by generating source code for the several different optimizations techniques used to optimize our code. The techniques used improved the efficiency by over 50%. With inlining and two issued firmware as well as all of the other optimization techniques used in the C portion of the code, the final assembly code had no store or load instruction and had 80 instruction in total. The original assembly code was composed of around 180 instructions, of which included 5 loads and 1 store.

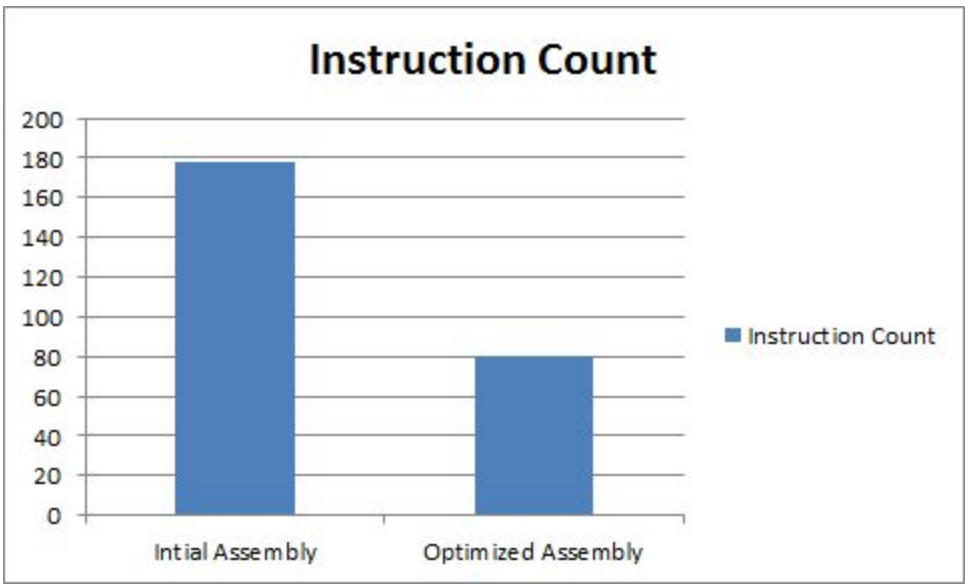

Figure 16. Instruction count for functional assembly code vs. final optimized assembly code.

4.2 Two Issued Slot Firmware:

Using two slot issued firmware instead of software, would be beneficial. There are many lines that could be run in parallel inside of the assembly code which would allow great savings in performance over a input sample. In our code there are a total of 15 instructions that can be run parallelly. Considering the inlined assembly code prior to two issued slot firmware was around 100 instructions lines this yielded almost 15% improvement in the final iteration of the code. Below is a partial picture of two issued slot in assembly.

```
mov r2, r0, lsr #8
cmp r2, #32768      subcs   r3, r2, #32768
movcs   r3, r3, lsr #3
addcs   r1, r3, #61440
movcs   r2, r1, lsr #12
addcs   r2, r2, #2
bcs .L3
cmp r2, #16384      subcs   r3, r2, #16384
movcs   r3, r3, lsr #2
addcs   r1, r3, #57344
movcs   r2, r1, lsr #12
addcs   r2, r2, #2
bcs .L3
cmp r2, #8192       subcs   r3, r2, #8192
movcs   r3, r3, lsr #1
addcs   r1, r3, #53248
movcs   r2, r1, lsr #12
addcs   r2, r2, #2
```

Figure 17. Partial assembly code showing implementation of two issued slots.

As seen in Figure 17 you can see that there are many lines were two issued slow firmware was not used. This is due to the number of registers used in the simulation being too small. Although this implementation would increase our performance the overall costs would outweigh the cost.

5 Conclusion

Audio compression considers a non-uniform method of quantization where steps are determined according to the human auditory system. Methods of uniform quantization are undesirable because humans are more likely to detect noise in lower level signals than higher level signals. Audio compression methods are then implemented using logarithmic functions as a result of how human responses to signal levels are. Two common methods are A-law and μ-law, with the latter being approximated piecewise in this project.

The piecewise approximation allows for an implementation of the algorithm using integer arithmetic instead of computing actual logarithms. We were able to do an initial implementation in C, which was optimized by reversing the order of piecewise interval lookups, using a lookup table to return log outputs, using constant return values in our algorithm, function inlining, and operator strength reduction. Further optimization was done at assembly level by reviewing the resulting compiled instructions from our C code. The assembly was optimized by reducing `str` and `ldr` operations, and allowing non-dependent instructions to run in parallel when considering two issued slot firmware. We can see that code optimizations in both C and at assembly level are greatly important when efficiency has high priority in implementation.

Audio compression methods are widely used throughout the world to reduce transmission bandwidth and storage of audio signals. Methods such as A-law and μ-law are effective because they are simple and designed to adhere to human auditory perception. As technology develops to better incorporate audio communications, we expect to see implementations with logarithmic functions at its core.

6 Bibliography

[1] SENG 440 Lesson 101 Project ‘Audio Compression’
http://www.ece.uvic.ca/~msima/TEACHING/COURSES/SENG_440/PROTECTED/SLIDES/SENG_440_slides_Lesson_project_101.pdf
[Last accessed July 30 2015]

[2] A-Law/Mu-Law Companding
http://www.young-engineering.com/docs/YoungEngineering_ALaw_and_MuLaw_Companding.pdf
[Last accessed July 30 2015]

[3] μ-law algorithm
https://en.wikipedia.org/wiki/%CE%9C-law_algorithm
[Last accessed July 30 2015]

Our source code can be found at:
https://github.com/trison/Audio_Compression