Project 4: Audio Compression and Decompression using 𝜇 Law

SENG 440: Final Report

Logo

Description automatically generated with medium confidence

July 30th, 2021

Faculty Supervisor: Miha Sima

|  |  |
| --- | --- |
| Team 8 | |
| V# | Name |
| V00868600 | Matt MacLeod |
| V00870522 | Calder Staude |

Table of Contents

[List of Figures ii](#_Toc78925370)

[List of Tables iii](#_Toc78925371)

[List of Equations iii](#_Toc78925372)

[1.0 Introduction 1](#_Toc78925373)

[2.0 Theoretical background 2](#_Toc78925374)

[3.0 Design Process 7](#_Toc78925375)

[3.1 Software Solution 7](#_Toc78925376)

[3.1.1 Software Architecture 7](#_Toc78925377)

[3.1.2 WAV file processing 9](#_Toc78925378)

[3.1.3 𝜇 Law Audio Compression Algorithm 11](#_Toc78925379)

[3.1.3 𝜇 Law Audio Decompression Algorithm 13](#_Toc78925380)

[3.2 Hardware Solution 15](#_Toc78925381)

[4.0 Software optimizations 21](#_Toc78925382)

[4.1 C Level Optimizations 21](#_Toc78925383)

[4.2 Lookup Table Implementation 22](#_Toc78925384)

[4.3 Reverse order of If statements 25](#_Toc78925385)

[4.5 Switch Statement 26](#_Toc78925386)

[4.6 Assembly Level Optimizations 28](#_Toc78925387)

[5.0 Performance 30](#_Toc78925388)

[5.1 Audio Compression 35](#_Toc78925389)

[5.2 Audio Decompression 36](#_Toc78925390)

[6.0 Results 37](#_Toc78925391)

[7.0 Conclusions 37](#_Toc78925392)

[8.0 References 39](#_Toc78925393)

[Appendix ###: Original (unoptimized) Assembly Pipelining 1](#_Toc78925394)

[Appendix ###: Optimized Assembly Pipelining 5](#_Toc78925395)

[Appendix E: VHDL Code for Audio Compression 7](#_Toc78925396)

[Appendix F: Switch Statement VS If Statement Optimizations 9](#_Toc78925397)

[Appendix XXX ASM for Unoptimized Solution 13](#_Toc78925398)

[Appendix XXX ASM for Optimized Reverse If Statement Solution (best solution) 20](#_Toc78925399)

# List of Figures

[Figure 1: 𝜇 Law Compression Logarithmic Curve 2](#_Toc78925897)

[Figure 2: 𝜇 Law Decompression Exponential Curve 3](#_Toc78925898)

[Figure 3: 𝜇 Law Input Value vs. Decompressed value. 4](#_Toc78925899)

[Figure 4: 𝜇 Law Straight Line Approximation [4] 5](#_Toc78925900)

[Figure 5: Software Architecture 8](#_Toc78925901)

[Figure 6: Microsoft RIFF standard 9](#_Toc78925902)

[Figure 7: Code snippet illustrating wave\_header\_t struct used to partition the wave data 10](#_Toc78925903)

[Figure 8: Code snippet showing the processing of the WAV formatting “sub chunk 1” 10](#_Toc78925904)

[Figure 9: Code snippet showing the processing and encoding of the data “sub chunk 2” 11](#_Toc78925905)

[Figure 10: Compression Algorithm Flow Chart 12](#_Toc78925906)

[Figure 11: UnOptimized 𝜇 Law Audio Compression 13](#_Toc78925907)

[Figure 12: Decompression Algorithm 14](#_Toc78925908)

[Figure 13: 𝜇 Law Decompression Code Snippet 14](#_Toc78925909)

[Figure 14: 𝜇 Law Decompression Code Optimized 15](#_Toc78925910)

[Figure 15: Waveform Propagation of Hardware Solution 16](#_Toc78925911)

[Figure 16: FPGA Synthesized Design Schematic 17](#_Toc78925912)

[Figure 17:Hardware Solution Routing 18](#_Toc78925913)

[Figure 18:Place and Routing of Logic Unit within FPGA 19](#_Toc78925914)

[Figure 19:Hardware Implementation on FPGA 20](#_Toc78925915)

[Figure 20:FPGA Utilization 20](#_Toc78925916)

[Figure 21:Look Up Table Flow Chart 23](#_Toc78925917)

[Figure 22: Execution Time of Software Solutions 35](#_Toc78925918)

[Figure 23: File Size of Software Solutions 36](#_Toc78925919)

[Figure 24: Resulting Wave File Compression Illustration 37](#_Toc78925920)

# List of Tables

[Table 1: Project Objectives and Contributions 4](#_Toc78924802)

[Table 2: Input Range and Chord Slope [1] 9](#_Toc78924803)

[Table 3: 𝜇 Law Binary Encoding [1] 10](#_Toc78924804)

[Table 4: 𝜇 Law Binary Decoding [1] 10](#_Toc78924805)

[Table 5: Examples of optimization techniques used in this solution: 25](#_Toc78924806)

[Table 6: Unoptimized Code Vs Optimized Code with Switch Statement 28](#_Toc78924807)

[Table 7: Optimized If Statement vs. Optimized Reverse If Statement 29](#_Toc78924808)

[Table 8: Optimized If Statement vs Optimized Switch Statement 30](#_Toc78924809)

[Table 9: Unnecessary Memory Store and Loads 32](#_Toc78924810)

[Table 10: Unnecessary Register Moves 32](#_Toc78924811)

[Table 11: Summary of the performance of the software solutions 33](#_Toc78924812)

[Table 12: Optimized If Statement Execution Time 34](#_Toc78924813)

[Table 13: Optimized Reverse If Statement Execution Time 35](#_Toc78924814)

[Table 14: Look-Up Table Execution Time 35](#_Toc78924815)

[Table 15: Switch Statement Execution Time 36](#_Toc78924816)

[Table 16: Unoptimized Execution Time 37](#_Toc78924817)

# List of Equations

[Equation 1: PCM to 𝜇 Law Function [1] 9](#_Toc78924842)

# 1.0 Introduction

North American telecommunications commonly use 𝜇 Law quantization for voice traffic audio compression and decompression [1]. By compressing the voice traffic, communication network devices can support greater volumes of traffic over a communication network. 𝜇 Law utilizes a logarithmic function to compress data. 𝜇 Law compression works by sacrificing the dynamic range of an input signal to allow the signal to be compressed. The compression is performed on a logarithmic curve leading to greater distortion to larger signals or louder sounds than smaller signals or quieter sounds. This method is effective because the human auditory system has a logarithmic transfer function and cannot distinguish distortions to louder noises as well as it can distinguish distortions to quiet noises. 𝜇 Law audio compression is designed to compress an input value of 14 bits to a compressed codeword of 8 bits. The compression factor of 𝜇 Law is 1.75 [2].

This project displays the software and hardware optimized implementation of 𝜇 Law audio compression and decompression for embedded systems. Optimizations have been specifically made for the S3C2440A 32-Bit CMOS Microcontroller [3]. The report features four software solutions and one hardware solution for 𝜇 Law compression. The breakdown of objectives and tasks composed by the team are shown in Table 1.

Table 1: Project Objectives and Contributions

|  |  |
| --- | --- |
| Objective | Contributor(s) |
| UnOptimized 𝜇 Law Software Solution | Calder Staude and Matt MacLeod |
| Operator Strength Reduction 𝜇 Law Software Solution | Calder Staude and Matt MacLeod |
| Optimized 𝜇 Law with If Statements Reverse Software Solution | Calder Staude and Matt MacLeod |
| Optimized 𝜇 Law with Switch Statement Software Solution | Calder Staude and Matt MacLeod |
| 𝜇 Law Lookup Table Software Solution | Calder Staude and Matt MacLeod |
| 𝜇 Law Hardware Solution | Calder Staude |
| Assembly Optimization | Calder Staude and Matt MacLeod |
| Performance Evaluation | Calder Staude and Matt MacLeod |
| Progress Report | Calder Staude and Matt MacLeod |
| Final Report Section 1, Introduction | Calder Staude |
| Final Report Section 2, Theoretical Background | Calder Staude and Matt MacLeod |
| Final Report Section 3, Design Progress | Calder Staude and Matt MacLeod |
| Final Report Section 4, Software Optimizations | Calder Staude and Matt MacLeod |
| Final Report Section 5, Performance | Calder Staude and Matt MacLeod |
| Final Report Section 6, Results | Calder Staude and Matt MacLeod |
| Final Report Section 7, Conclusion | Matt MacLeod |
| Final Report Formatting | Matt MacLeod |
| Project Presentation Slides | Calder Staude and Matt MacLeod |

# 2.0 Theoretical background

The audio compression is implemented using a logarithmic curve while the decompression is implemented using an exponential curve. The logarithmic and exponential curves are shown in Figure 1 and Figure 2 respectively. Figure 3 shows the input value compared to the decompressed output. The figures are derived from the Optimized 𝜇 Law with If Statements Reverse Software Solution.

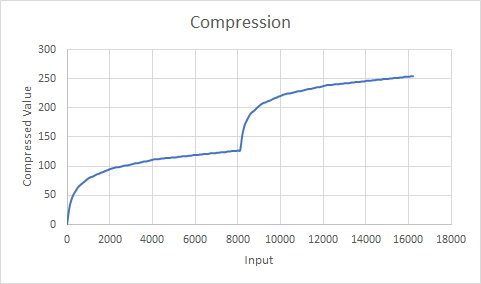


Figure 1: 𝜇 Law Compression Logarithmic Curve

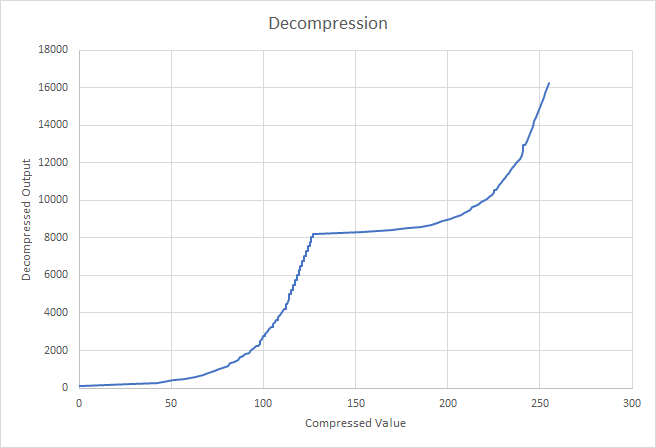


Figure 2: 𝜇 Law Decompression Exponential Curve

Chart, line chart

Description automatically generated

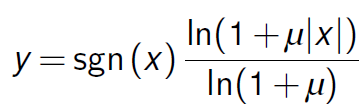
Figure 3: 𝜇 Law Input Value vs. Decompressed value.

From Figure 3, one may see the error between the input value and the decompressed value, noted by the squiggly line segments in the input range of 4000 to 8000 and 12000 and 16000. It is because the ranges correspond to the larger amplitude input signals. During 𝜇 Law compression, only the most significant five bits are preserved, and the remaining 9 least significant bits are discarded allowing for the input signal to be compressed. The discarding of bits leads to a loss of information and an error between the input and decompressed signal ultimately distorting the audio.

The reason the error segments are in the range of 4000 to 8000 and 12000 and 16000 is because these are the extremes of the input signals. The 14th bit is the input’s sign, and the remaining 13 bits are the signal’s magnitude. From the provided documentation a 0 in the 14th bit position indicates a positive and a 1 indicates a negative. Using this convention, the greatest positive value is 8190 in decimal or 01 1111 1111 1111 in binary. The largest negative value is -8191 in decimal or 11 1111 1111 1111 in binary.

To implement 𝜇 Law, the team has chosen to utilize a Logarithmic Pulse Code Modulation (PCM). A logarithmic PCM was chosen over a uniform PCM because it is preferable to have a larger error for a signal with a large amplitude than a signal with a small amplitude. It is preferable to have this error because the human auditory system has a logarithmic transfer function. As a result of this function, the system will not detect distortions to audio sounds with a large amplitude as well as it will detect distortions to signals with a small amplitude. By sacrificing information in the form of discarding the bits following the most significant 5 bits, the algorithm may achieve compression. As a result of using a logarithmic PCM, the quantization step will be greater for large signals than smaller signals. The large quantization step will lead to greater distortions to large signals. Additionally, the large quantization step will allow for a compression factor of 1.75.

To implement 𝜇 Law, the team has used Equation 1 to perform PCM to 𝜇 Law.



Equation 1: PCM to 𝜇 Law Function [1]

The argument *x* is bound to 0 ≤ |*x*| ≤ 1 and the parameter μ is bound to 0 ≤ *μ* ≤ 255. When *μ* = 0 there is no compression and when *μ* = 255 maximum compression is performed. Today it is standard to use maximum compression (*μ* = 255) [1].

To implement 𝜇 Law with maximum compression, one may approximate the logarithmic curve of 𝜇 Law with 8 straight-line segments. Figure # shows the approximated curve. These line segments are chords.

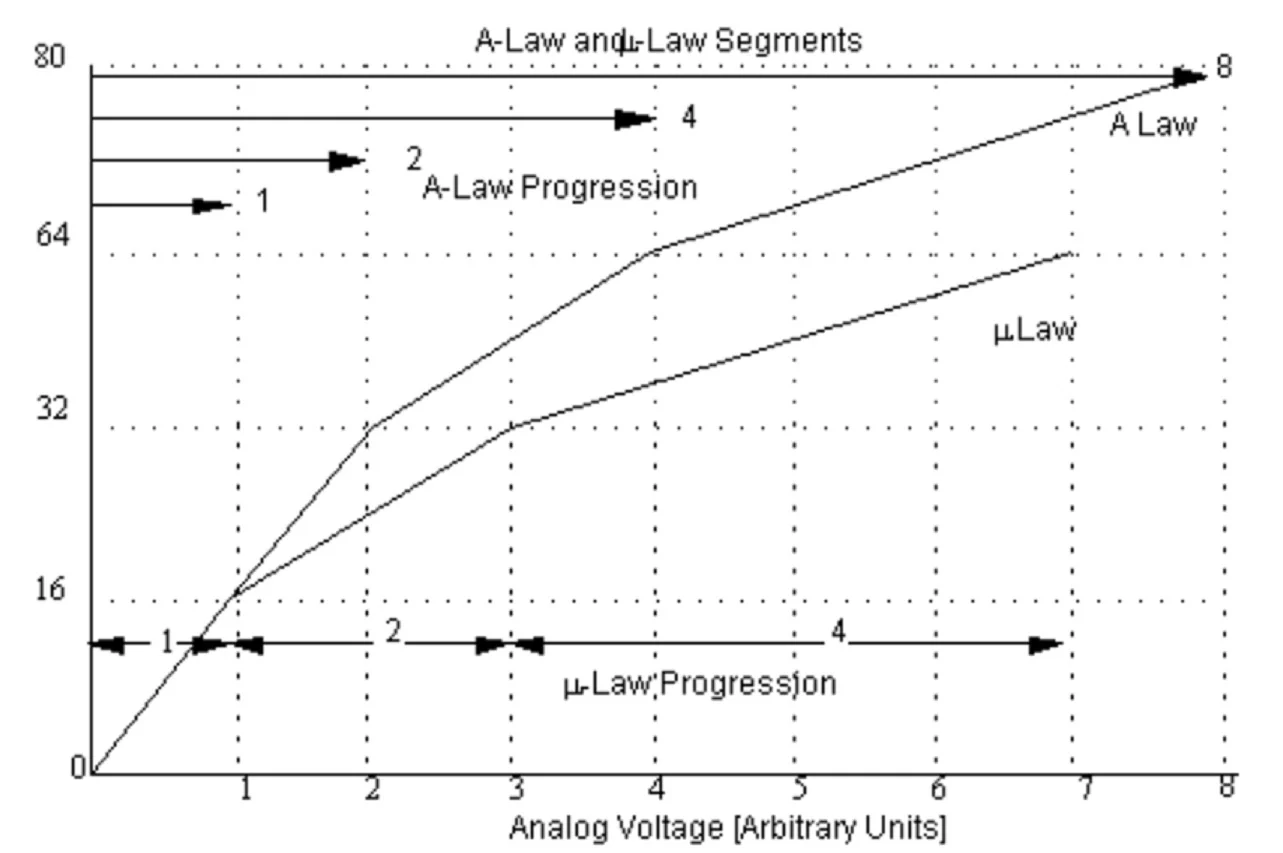
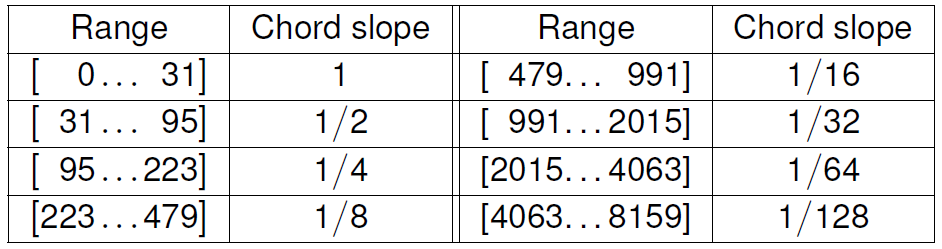


Figure 4: 𝜇 Law Straight Line Approximation [4]

The slope of each chord is half the previous chord. The table for the Chord slope and range of inputs may be seen in Table 2.

Table 2: Input Range and Chord Slope [1]



𝜇 Law compression works by receiving a 14-bit input. The input’s sign and magnitude are then determined. The sign is kept as the most significant bit of the compressed codework and the most significant bit of the decompressed word. If the input is positive the sign is 0 if the input is negative the sign is 1. From the input’s magnitude the signal’s Step and Chord is determined. The Step is the four bits following the first one in the input’s 13-bit magnitude. The Chord is determined by how large the input value is. The larger the value the smaller the Chord value. See Table 3 for a 𝜇 Law binary encoding table. See Table 4 for a 𝜇 Law binary decoding table.

Table 3: 𝜇 Law Binary Encoding [1]

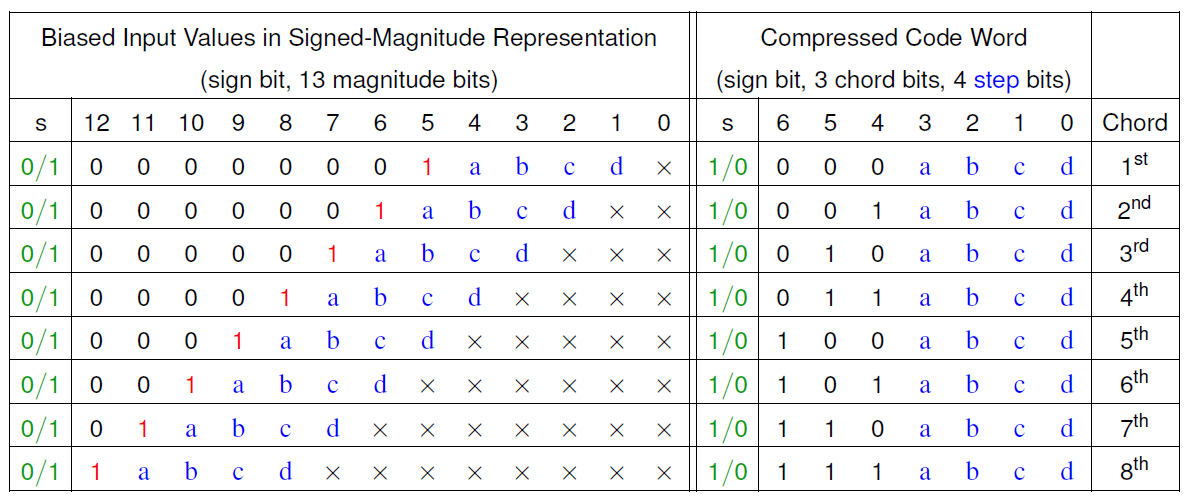
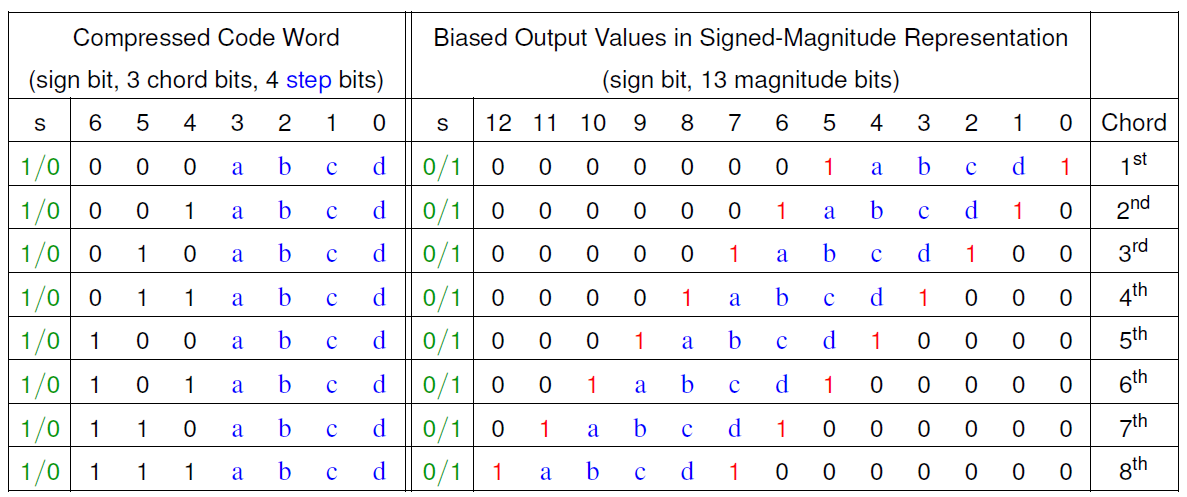


Table 4: 𝜇 Law Binary Decoding [1]



# 3.0 Design Process

## 3.1 Software Solution

The full source code is available to reference online: <https://github.com/macleod-matt/Audio-Compression-Decompression-Embedded>

In addition to appendix A below.

### 3.1.1 Software Architecture

Below shows a high-level description of the software architecture used in this project:

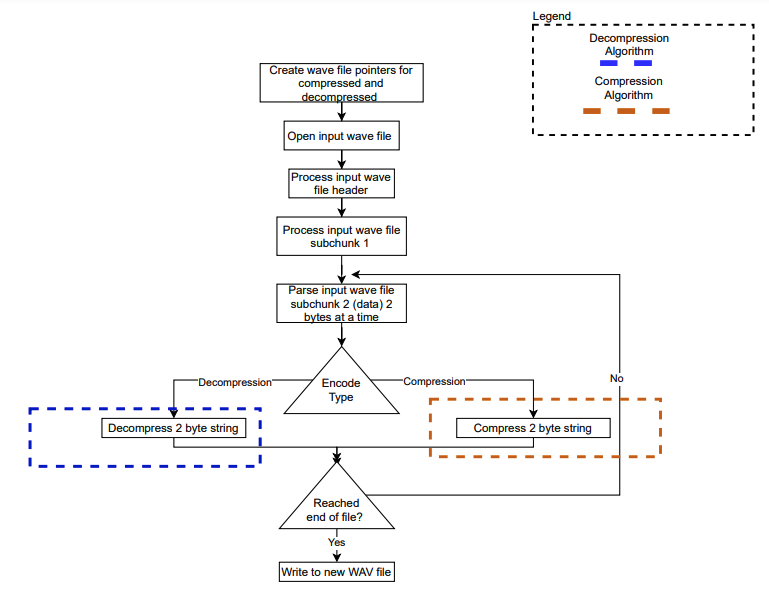


Figure 5: Software Architecture

The above figure shows a high-level description of the architecture used to process a .WAV file and then perform the compression and then decompression to reduce the size and then recover the sound of the file. As shown in Appendix Abelow, incorporating the source code. This solution was modeled in a functional programming implementation that was designed to simplify the API responsible for processing of the WAV file as much as possible such that fully optimized compression and decompression algorithms could be achieved, while minimizing the time and complexity required to develop a reduced wave file processing structure.

Another potential software architecture that was investigated was a finite state machine FSM. This solution would switch between the following states: (Process Sate, Compress State, Write State, Decompress State). This architecture, although would have been feasible, was ultimately considered to be unnecessarily complicated for this project due to the memory management complexity associated with formatting the wave file data as structs/nodes for a linked list so that the data position in the WAV file and formatting parameters could be handled as they are encoded for compression/decompression and then written into the new file during the state transitions.

### 3.1.2 WAV file processing

A low-level method had to be developed to process and then encode the input WAV files. The Method used in this solution involved being very deliberate in reading and processing the bits individually associated with the 3 main subcomponents called “chunks” that comprised the data format of the Microsoft RIFF standard for header information, formatting and data as shown in Figure 6 below:

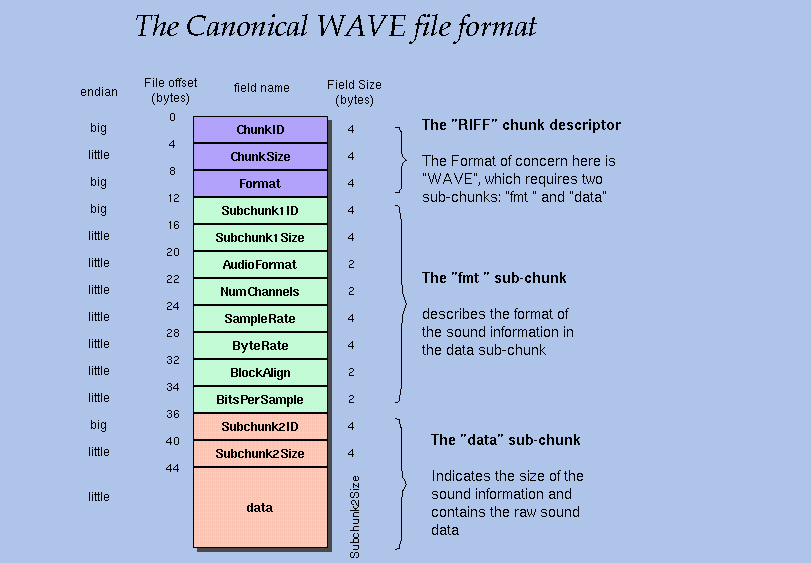


Figure 6: Microsoft RIFF standard

As shown in the code snippet below from the API read\_wav.h, the data type “wave\_header\_t” was created to save the data as it was read from each call to fread() so that it could be partition into the different sections as required to create a readable WAV file.

As shown in Appendix Bbelow. The data of the WAV was opened and then stored in a pointer to each sub chunk partition of the RIFF file structure.

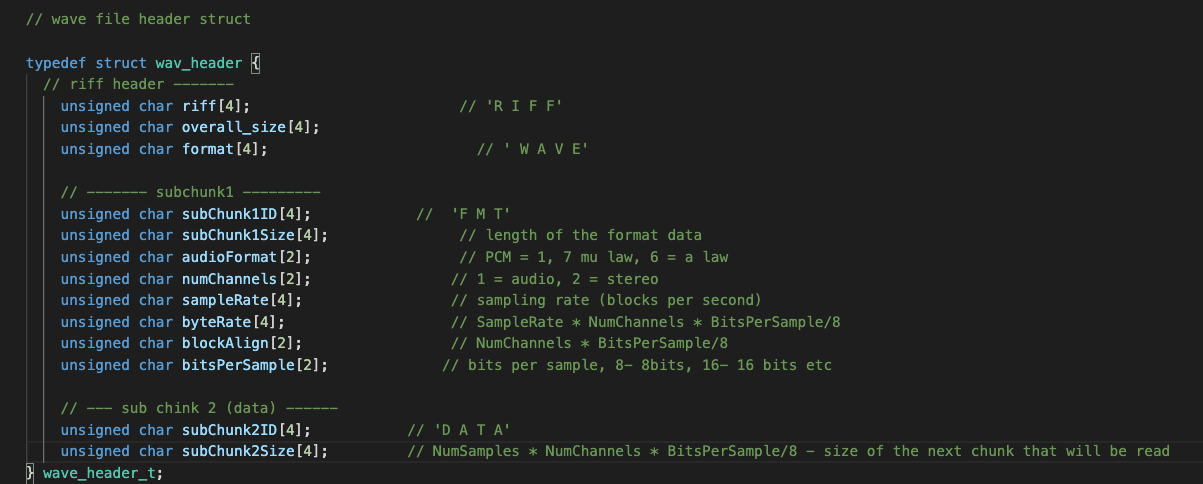


Figure 7: Code snippet illustrating wave\_header\_t struct used to partition the wave data

As mentioned above, another solution to accomplish this would have been to incorporate an FSM to subjugate the transitioning between the processing of the input file, the encryption and then the writing of the compressed/decompressed data to another WAV file. However, the approach taken in the proposed solution was far more simplistic, being deliberate in writing each byte of data as it is written and then formatting the output file right away made the processing of the WAV data formatting much (shown in Figure 8 below):

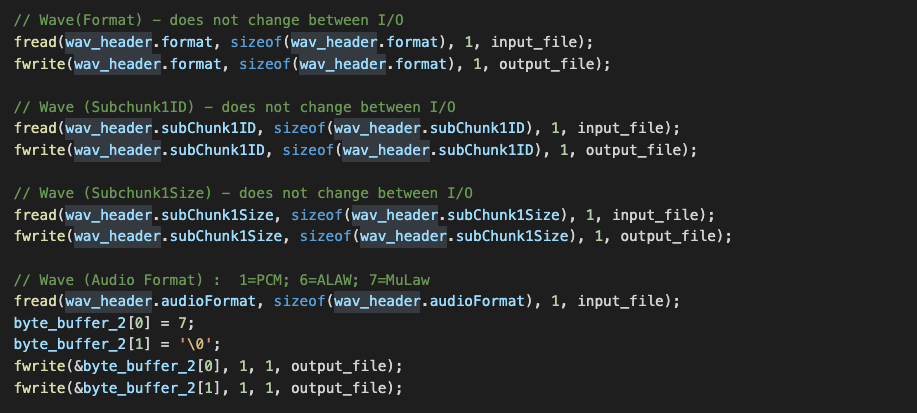


Figure 8: Code snippet showing the processing of the WAV formatting “sub chunk 1”

According to the WAVE PCM sound file format standard documentation [5] the first 36 bytes that are read are associated with the file formatting. Notable parameters such as audio format, number of channels were deliberately hard coded as each WAV file that is processed using this algorithm will be consistent with one another.

The remaining bytes to be processed are the data (as shown in Figure 6 above). The overall size of the file is read once the file is first opened and then saved as a counter variable. As the algorithm processes the descriptor and then “sub chunk 1”, the counter variable is decremented by the number of bits that are processed. Once the algorithm reaches “sub chunk 2” (the data to be encoded), the counter variable is used as the bound of the for loop shown in the code snippet below depending on the type of encoding (compressed/decompressed):



Figure 9: Code snippet showing the processing and encoding of the data “sub chunk 2”

Finally, the encoded data is written to the output file and the files are closed.

### 3.1.3 𝜇 Law Audio Compression Algorithm

The 𝜇 Law Compression Algorithm works by taking an input signal of 13 bits with the first bit being a sign bit. The input is entered into a series of if statements used to identify the chord of the audio input. These if statements are shown as diamonds in Figure 10. Figure 10 shows the Original (unoptimized) code for implementing 𝜇 Law. If the input matches a chord, the four most significant bits will be kept as the step. The remaining bits will be truncated and lost allowing for data compression. The output of the algorithm will be a 7-bit value with the most significant bit being the sign, the following three bits will be the chord and the last least significant bits will be the step.

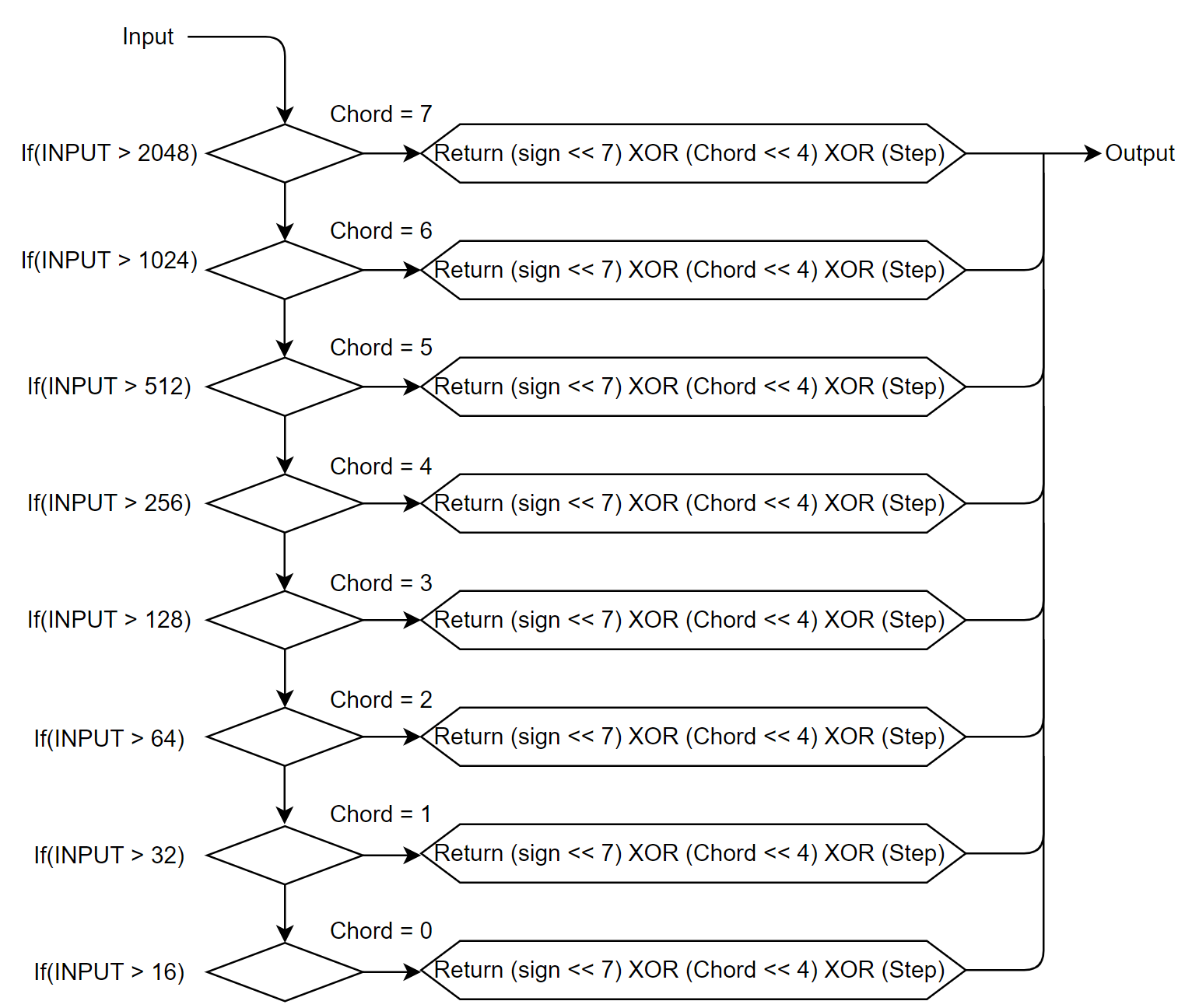


Figure 10: Compression Algorithm Flow Chart

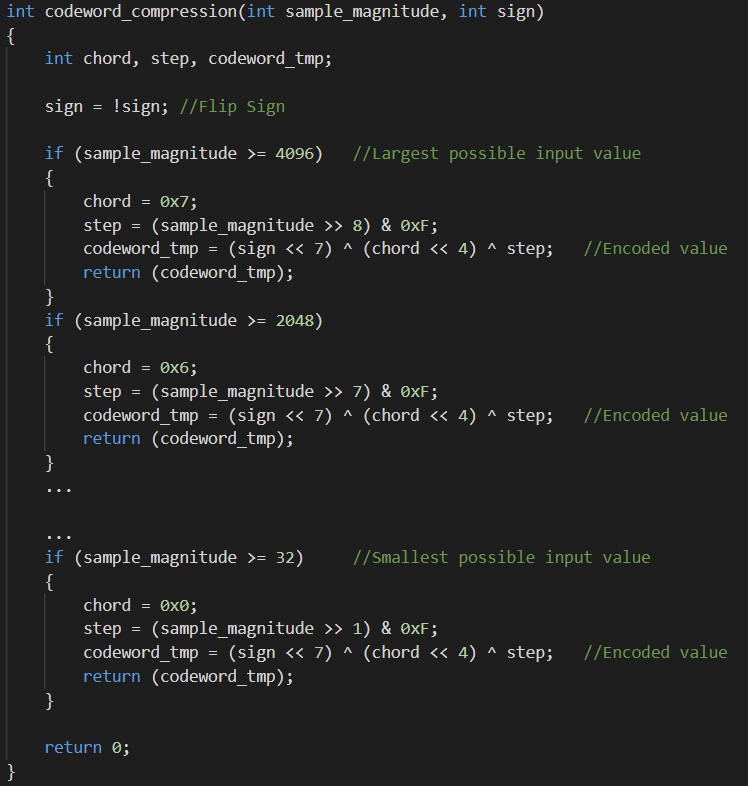


Figure 11: UnOptimized 𝜇 Law Audio Compression

The original (unoptimized) code shown in Figure 11 is used as the basis when comparing optimized software revisions.

### 3.1.3 𝜇 Law Audio Decompression Algorithm

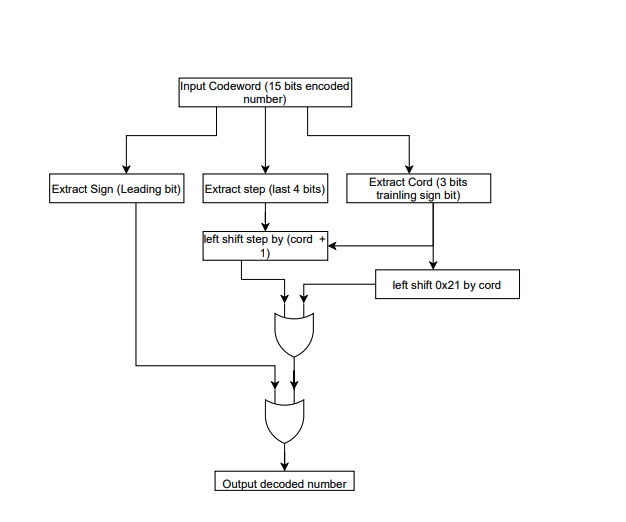


Figure 12: Decompression Algorithm

As shown in Figure 12 above, the decompression algorithm that was implemented to recover the original 15 bit integer from the compressed audio file. This algorithm was implemented in software and can be realized in muLawAPI.c.

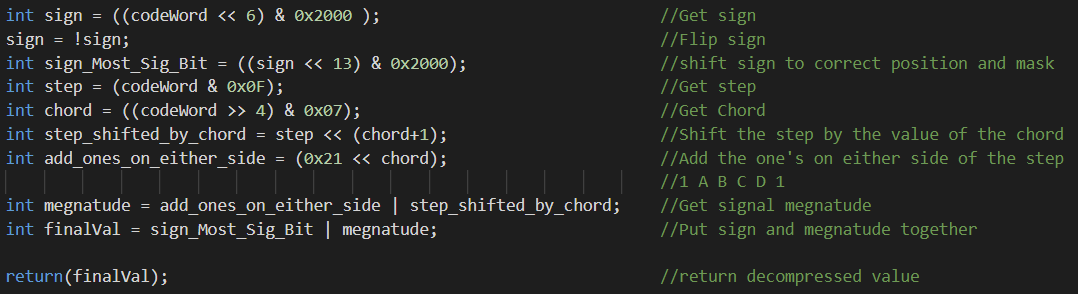


Figure 13: 𝜇 Law Decompression Code Snippet

The 𝜇 Law decompression was summarized into the following 9 steps.

1. Determine the signals sign by shifting the compressed word by 6 bits and masking by 0x2000 in hex.
2. Flip the sign. 𝜇 Law requires flipping the sign of the input value to perform signal compression, thus, the sign must be flipped back during decompression.
3. After the sign is determined, it is shifted to its original value as the most significant bit.
4. The step is computed by masking the code world by 0xF. This isolates the least significant four bits of the compressed value holding the step.
5. The chord is found by shift the compressed signal four bits to the right, eliminating the step. The value is then masked by 0x7 to exclude the sign and isolate the chord.
6. The step is then shifted to the left by the value of the chord plus one. This puts the step in the correct spot and makes room for the step to have one’s on either side as outlined in the 𝜇 Law decompression table.
7. Ones are added to either side of the step by using the OR operator and the value of 33 in decimal or 0x21 in hex.
8. The decompressed signal magnitude is then found by using the OR operator with the step.
9. Finally, the sign is combined with the signal's magnitude by using the OR operator to return the decompressed signal value.

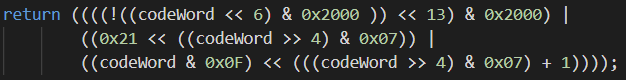


Figure 14: 𝜇 Law Decompression Code Optimized

To optimize the process of decompression, the 9 steps outlined above can be written into one line of code, eliminating the need to store values as local variables. Because the input value and the decompressed value are different, there is an audible difference between the input audio and in the decompressed audio output.

## 3.2 Hardware Solution

The hardware solution involves programming and FPGA in VHDL code and simulating the code. The hardware solution logic is similar to that of the software solution. To compress an audio signal multiple statements are used to determine how many leading zeros the signal has. Once the if statement is entered, the signal is broken up into the step, chord, and sign. These parts are then put together to form the compressed output signal.

Signal decompression works by shifting the step by the magnitude of the chord. The sign is then added, and the decompressed signal is produced. The VHDL code is shown in Appendix D.

Figure 15 shows the simulated performance of the Hardware Solution. One may see that the FPGA is able to perform the operation almost instantaneously. This is possible because the digital components used to implement the design do not consist of items that require latching signals. On the right side of the figure, one may see that the input signal is 0173 in hex and that the output is the chord, which is 5 and the step is 4. Both these values are used in the audio compression.

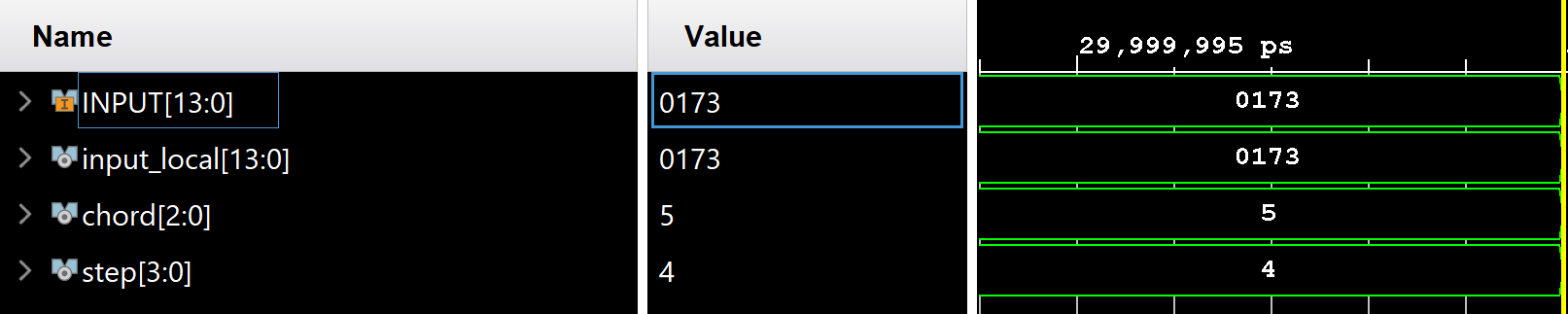


Figure 15: Waveform Propagation of Hardware Solution

Figure 16 shows the synthesized designed schematic that uses flip flops to act like if statements allowing the FPGA to perform 𝜇 Law Audio compression.

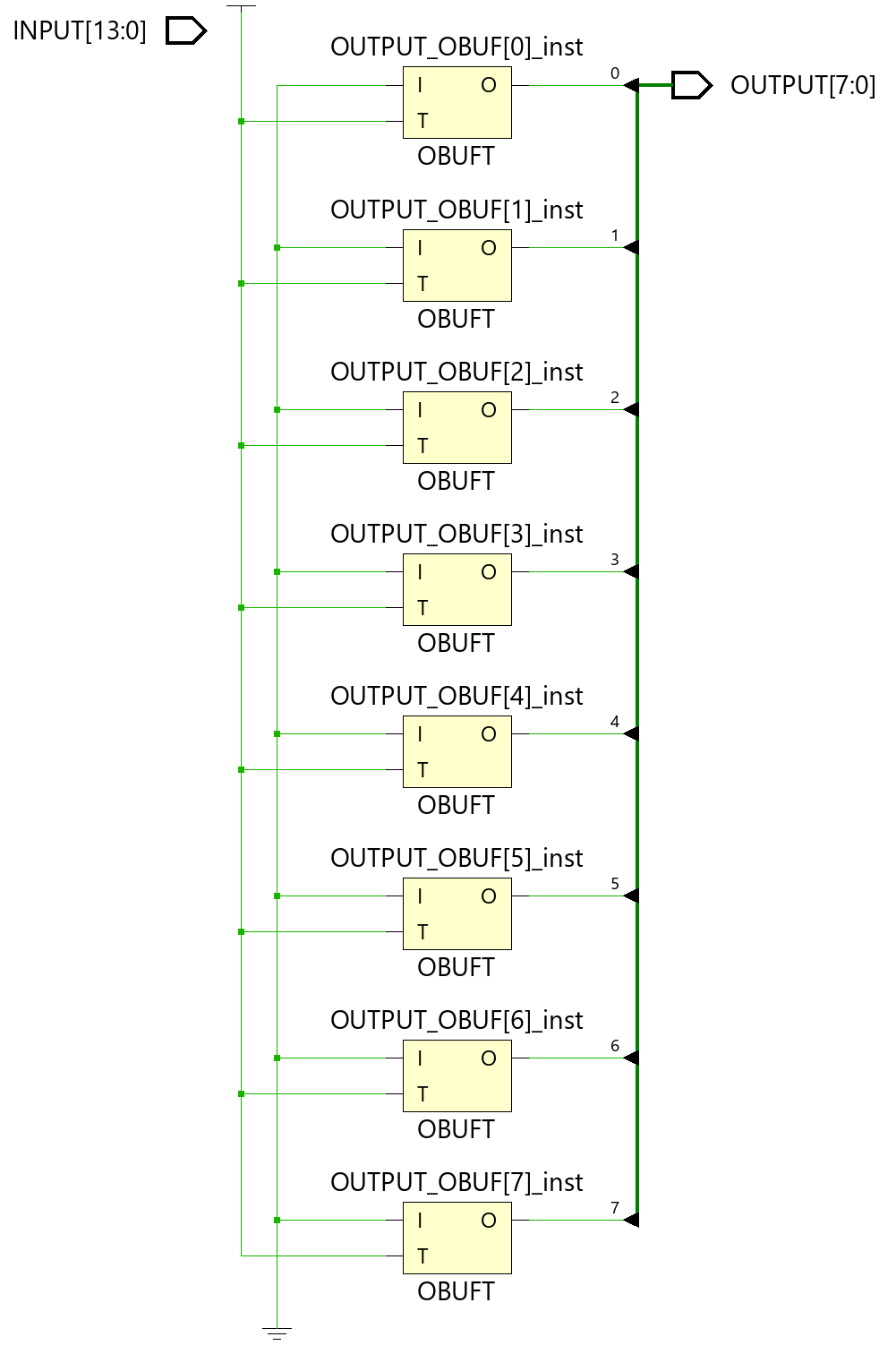


Figure 16: FPGA Synthesized Design Schematic

Figure 17 and Figure 18 show the place and routing used to route the logic through the FPGA and implement the solution.  Figure 17 shows the entire FPGA.  Figure 18 shows a single logic block within the FPGA.

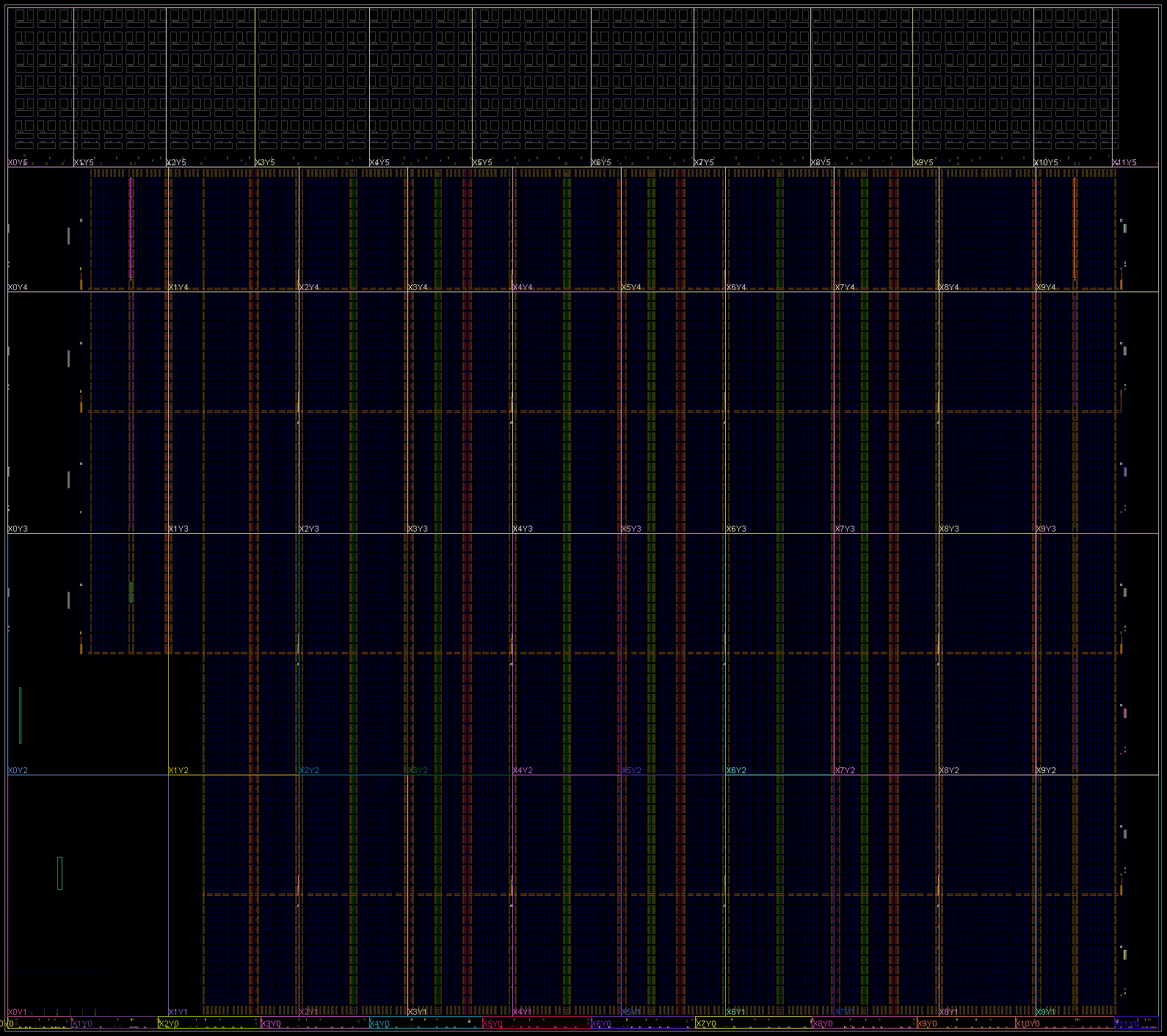


Figure 17:Hardware Solution Routing

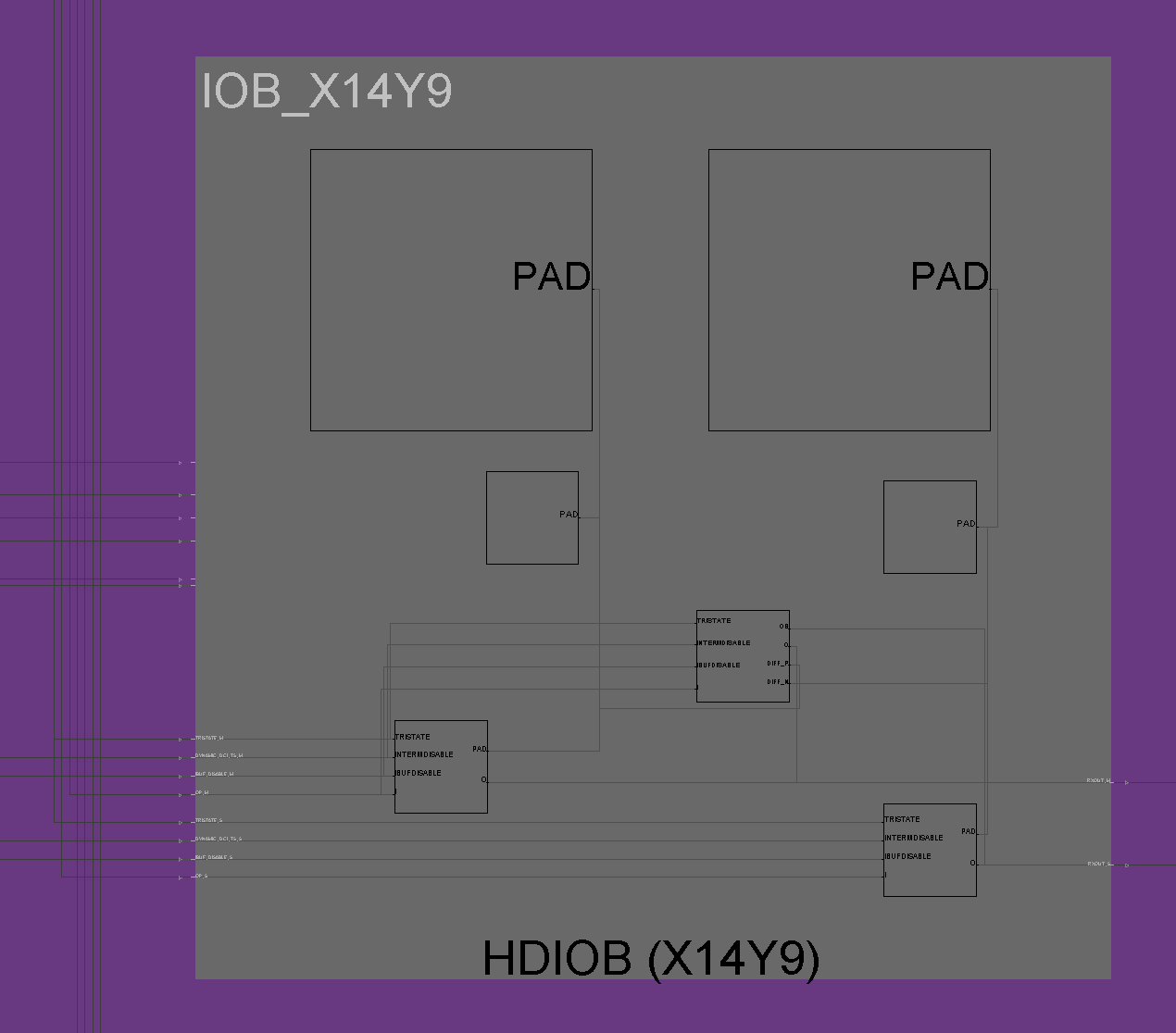


Figure 18:Place and Routing of Logic Unit within FPGA

Figure 19 shows the digital logic used in the FPGA to implement the design.  The design appears as a line because the design is using flip-flops in parallel to determine the cord of the input signal.  Once the chord is determined bits are shifted to determine the step and the outcome is produced.  Because there are very few dependencies the hardware, the FPGA can perform the computation very quickly.  Theoretically it can perform it in a single clock cycle which is ten nano seconds due to the board having a 100 MHz clock frequency.

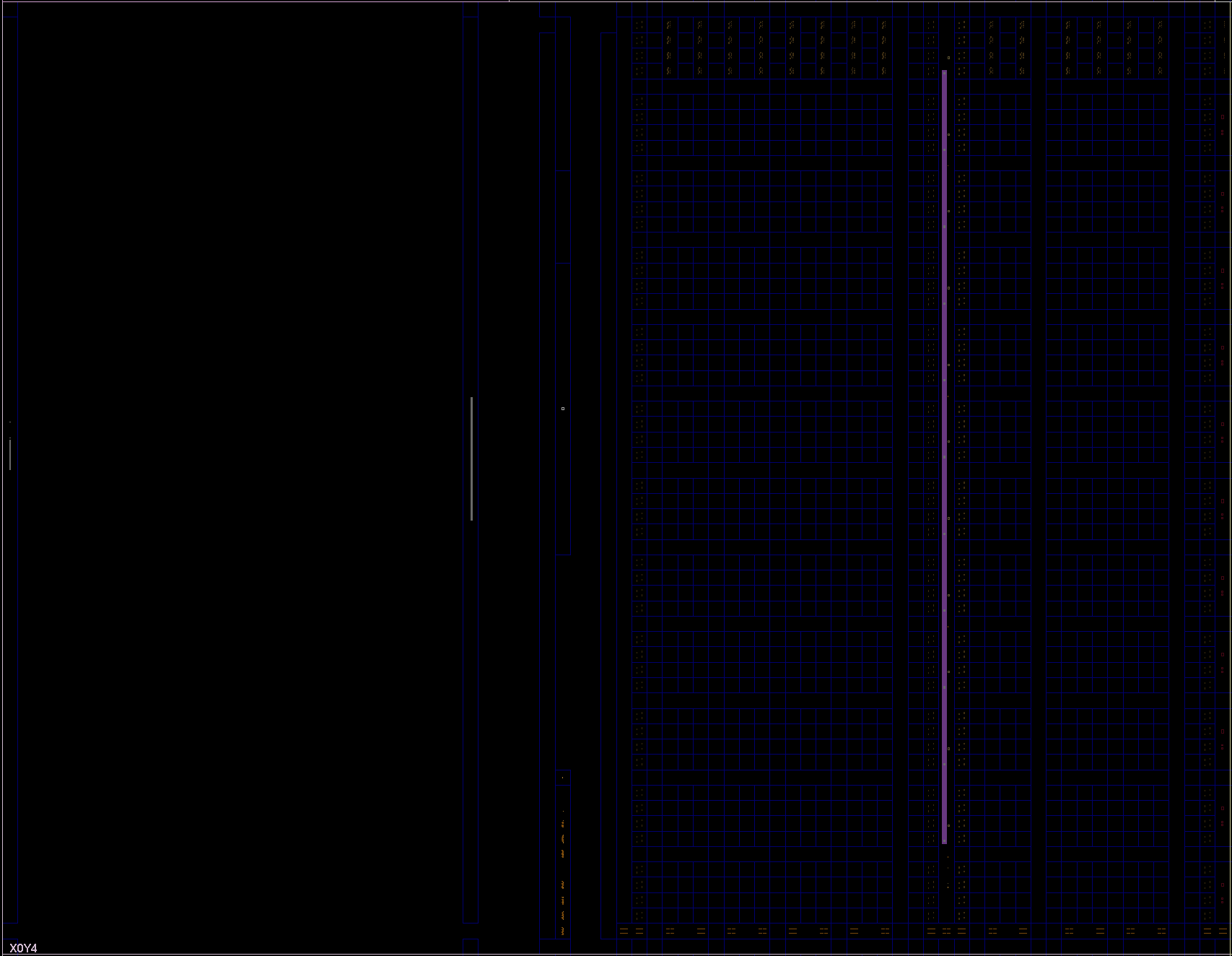


Figure 19:Hardware Implementation on FPGA

Figure 20 shows the FPGA Utilization. Because the design consists of only a few digital components, the hardware implementation only utilized 2% of the FPGA. Because this utilization is so little, one may want to consider using a smaller and less expensive FPGA than the one selected for the prototype.

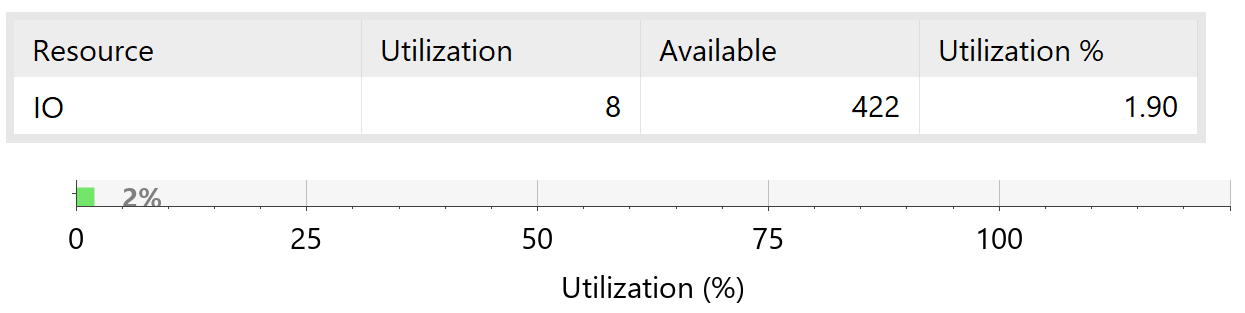


Figure 20:FPGA Utilization

# 4.0 Software optimizations

## 4.1 C Level Optimizations

With the functional architecture that was chosen. There were not operator level optimizations that could have been made that would have resulted in a significant difference in runtime or compile time speed. Instead, only minor improvements were made such as operator strength reduction and loop unrolling to improve the run time of the program. Additionally, we intentionally made use of preprocessor commands for both test cases and to improve the overall performance and cleanness of the program.

Table 5 below summarizes the c level optimizations that were made in this solution:

Table 5: Examples of optimization techniques used in this solution:

|  |  |  |
| --- | --- | --- |
| Technique | Code Pre-Optimization Example | Code Post-Optimization Example |
| Operator Strength Reduction | int sign = ((codeWord << 6) & 0x2000 );  sign = !sign;  int sign\_Most\_Sig\_Bit = ((sign << 13) & 0x2000); //shift sign to correct position and mask  int step = (codeWord & 0x0F); //Step  int chord = ((codeWord >> 4) & 0x07); //chord  int step\_shifted\_by\_chord = step << (chord+1);  int add\_ones\_on\_either\_side = (0x21 << chord); // add the 1 A B C D 1 33 dec, 100001 bin, 21 hex  int megnatude = add\_ones\_on\_either\_side | step\_shifted\_by\_chord;  int finalVal = sign\_Most\_Sig\_Bit | megnatude; | int finalVal =  ((((codeWord << 6) & 0x2000 ) << 13) & 0x2000) |  ((0x21 << ((codeWord >> 4) & 0x07)) |  ((codeWord & 0x0F) << (((codeWord >> 4) & 0x07) + 1))); |
| Loop unrolling | int16\_t data ;  Int16\_t codeword;  for (int i =0; i < overall\_sizel; i++){  Data =bytes\_to\_int16(input\_data\_buffer[i], inputfile\_data\_buffer[i] );  Codeword = codeword\_compression(data, signum(data));  output\_file\_Data \_buffer[i /2] = codeword;  } | int16\_t data1, data2,data3;  for (int i =0; i < overall\_sizel; i+=8){  data1=bytes\_to\_int16(input\_data\_buffer[i +1], inputfile\_data\_buffer[i] );  data2=bytes\_to\_int16(input\_data\_buffer[i +2], inputfile\_data\_buffer[i + 3] );  Data1=bytes\_to\_int16(input\_data\_buffer[i +4], inputfile\_data\_buffer[i + 5] );  Data1=bytes\_to\_int16(input\_data\_buffer[i +6], inputfile\_data\_buffer[i + 7] );  Codeword1 = codeword\_compression(data, signum(data));  Codeword2 = codeword\_compression(data, signum(data));  Codeword3 = codeword\_compression(data, signum(data));  Codeword4 = codeword\_compression(data, signum(data))  output\_file\_Data \_buffer[i /2] = codeword1;  output\_file\_Data \_buffer[i /2 +1] = codeword2;  output\_file\_Data \_buffer[i /2 + 2] = codeword3;  output\_file\_Data \_buffer[i /2 + 3] = codeword4;  } |
| Variables Data Types | Int cord, step;  Int codeword\_temp; | Register int cord, step, codeword\_temp; |
| Macro Implementation | Int magnitude (int sample){  if(sample < 0){  Return -sample;  } else{  Return sample;  }  } | #define magnitude(sample) ((sample < 0) ? -sample : sample) |
| Lookup table | See Section 4.1.2 Below | |

## 4.2 𝜇 Law Lookup Table Software Solution

A lookup table was created to implement 𝜇 Law.  The lookup table maps every possible input to a return value using a switch statement.  By not requiring any computation to perform 𝜇 Law compression, the software solution has a small execution time compared to other solutions.

The drawback of using a lookup table is that it requires a large amount of memory to store.  If the lookup table is too large to be stored in cache, the processor will be required to fetch data from memory.  Cache misses and memory fetches are extremely costly for execution time and stall the processor’s pipeline.  The lookup table used in the project has a file size of 958,712 bytes which is 130 times larger than the second largest software solution’s file size.  The look up table’s performance and file size is discussed further in section 5.0 of the report. While the SENG 440 lab ARM processor was able to implement the lookup table, the lookup table may not be able to be used by other embedded devices due to its large memory requirement.  The lookup table solution’s performance and file size relative to other solutions is detailed further in section 5 of the report.  Figure 21 shows a high-level description of the lookup table algorithm.

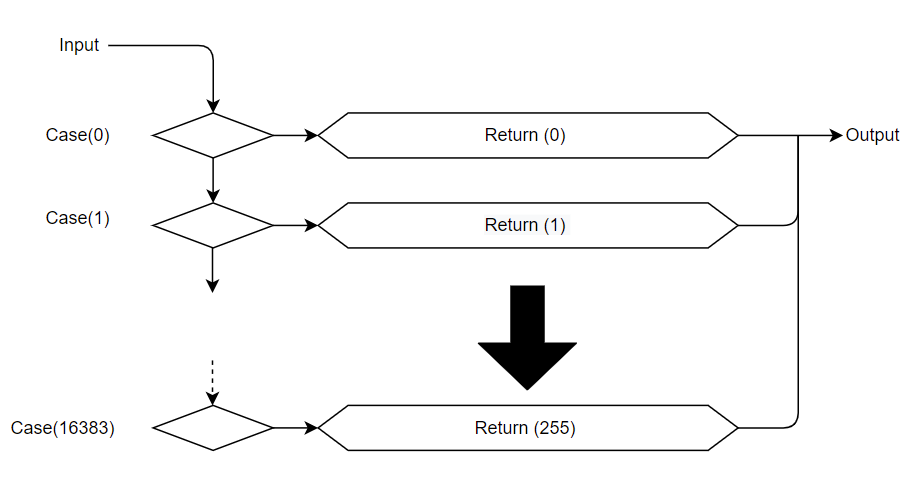


Figure 21:Look Up Table Flow Chart

See below code snippet of the lookup table implementation:

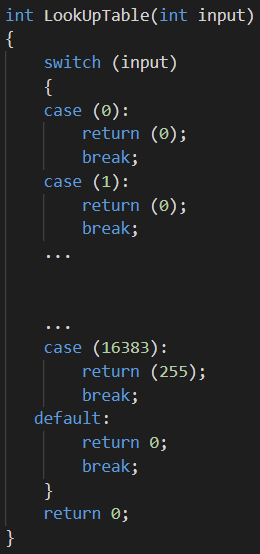


Figure #: Lookup Table Sample Code

Table 6: Unoptimized Code Vs Optimized Code with Switch Statement

|  |  |
| --- | --- |
| Original Code (unoptimized) with If Statements | Optimized Code with Lookup Table |
| int codeword\_compression(int sample\_magnitude, int sign)  {  int chord, step;  int codeword\_tmp;  sign = -sign; //Flip Sign    if (sample\_magnitude & (1 << 12))  {  chord = 0x7;  step = (sample\_magnitude >> 8) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  if (sample\_magnitude & (1 << 11))  {  chord = 0x6;  step = (sample\_magnitude >> 7) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ...  ...  if (sample\_magnitude & (1 << 5))  {  chord = 0x0;  step = (sample\_magnitude >> 1) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  return 0;  } | int LookUpTable(int input)  {  switch (input)  {  case (0):  return (0);  break;  case (1):  return (0);  break;  ...    ...  case (16383):  return (255);  break;  default:  return 0;  break;  }  return 0;  } |

## 4.3 Optimized 𝜇 Law with If Statements Reverse Software Solution

To further optimize the software solution, the if statements from the Operator Strength Reduction 𝜇 Law Software Solution were ordered from the largest input value to the smallest input value. The Optimized 𝜇 Law with If Statements Reverse Software Solution uses much of the same code as the Operator Strength Resection code, however, the ordering of the if statements is reversed. Table 7 shows the changes in the code.

Table 7: Optimized If Statement vs. Optimized Reverse If Statement

|  |  |
| --- | --- |
| Optimized If Statement | Optimized Reverse If Statement |
| ...  if (sample\_magnitude & (1 << 12))  {  chord = 0x7;  step = (sample\_magnitude >> 8) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ...  …  if (sample\_magnitude & (1 << 5))  {  chord = 0x0;  step = (sample\_magnitude >> 1) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ... | ...  if (sample\_magnitude & (1 << 5))  {  chord = 0x0;  step = (sample\_magnitude >> 1) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ...  ...  if (sample\_magnitude & (1 << 12))  {  chord = 0x7;  step = (sample\_magnitude >> 8) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ... |

After comparing the execution times of the Operator Strength Reduction 𝜇 Law Software Solution to the Optimized 𝜇 Law with If Statements Reverse Software Solution, it was determined that the If Statement Reversed solution had an execution time 17% lower than the Operator Strength Reduction Solution. The difference in performance is likely because the audio file that the solutions were tested on mainly consisted of silence and quiet sounds. Quiet sounds require less bits to represent in the audio file. When the software solutions read through the audio the Operator Strength Reduction Solution begins its if statements looking for large values before moving to smaller values. The optimized, If Statement Reversed Solution does the opposite, first looking for small values before continuing to larger values. By looking for quiet sounds first, the if statement branches are taken sooner and the software runs faster.

## 4.5 Switch Statement

To further optimize the code, an alternative software solution was considered. In the alternative software solution, the if statements in the Operator Strength Reduction Solution were replaced by a switch statement. The challenge with creating the Switch Statement solution is that a switch statement requires a fixed input value unlike if statements that support condition checks. To accommodate this requirement the conditional check is processed before the switch statement. The fixed integer value for the switch statement is determined by iterating through a while loop. In each iteration a counter value named “NumberOfBits” is increased while the input value is shifted one bit to the right. The loop exits once the input bits are shifted to zero. The counter value is then used as an input to the switch statement. A sample of the switch statement code may be seen in table 8.

Table 8: Optimized If Statement vs Optimized Switch Statement

|  |  |
| --- | --- |
| Optimized If Statement | Optimized Switch Statement |
| ...  if (sample\_magnitude & (1 << 12))  {  chord = 0x7;  step = (sample\_magnitude >> 8) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ...  …  if (sample\_magnitude & (1 << 5))  {  chord = 0x0;  step = (sample\_magnitude >> 1) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  return (codeword\_tmp);  }  ... | ...  while (sample\_magnitude > 0)  {  sample\_magnitude = sample\_magnitude >> 1;  NumberOfBits++;  }  switch (NumberOfBits)  {  case 14:  chord = 0x7;  step = (sample\_magnitude >> 8) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  debug\_print("chord: %d, step: %d, codeword\_tmp: %d | ", chord, step, codeword\_tmp);  return (codeword\_tmp);  break;  ...    ...  case 7:  chord = 0x0;  step = (sample\_magnitude >> 1) & 0xF;  codeword\_tmp = (sign << 7) ^ (chord << 4) ^ step;  debug\_print("chord: %d, step: %d, codeword\_tmp: %d | ", chord, step, codeword\_tmp);  return (codeword\_tmp);  break;  }  ... |

## 4.6 Assembly Level Optimizations

After compiling the Optimized 𝜇 Law with If Statements Reverse Software Solution, it was evident additional optimizations could be applied to the assembly. Looking at the assembly for the compression and decompression, one can see optimizations made towards the compression also apply for optimizations made for the decompression. One such optimization is removing instructions that would store data and then immediately load the same data from memory. This occurred in both the assembly for the compression and decompression. Attempts were made at the C level to help the compiler not include these errors by having all five valuables used in the compression function kept in registers. Despite making this adjustment to the C code, the compiler continued to insist on making unnecessary accesses to memory. As a result, the optimizations must be applied in the assembly. This is a significant Optimization as load and store operations are expensive and require many processor cycles to perform. The Unoptimized and Optimized assembly may be seen in Table 9.

Table 9: Unnecessary Memory Store and Loads

|  |  |
| --- | --- |
| Original Assembly | Optimized Assembly |
| str r0, [fp, #-24]  str r1, [fp, #-28] ;Remove  ldr r1, [fp, #-28] ;Remove  cmp r1, #0 | str r0, [fp, #-24]  cmp r1, #0 |

Further Optimizations were made to the assembly by removing a register move operation. As shown in Table 10, the instruction, mov r0, r3, that moves the contents of register 3 to register 0 can be implemented in the load instruction above, ldr r3, [fp, #-20] by loading data into register 0 directly. This adjustment will remove a processor cycle allowing for the program to have a smaller execution time. The optimization of replacing the move operation by directly writing to the destination register was found in both the compression and decompression assembly. An example showcasing this optimization may be seen in Table 10.

Table 10: Unnecessary Register Moves

|  |  |
| --- | --- |
| Original Assembly | Optimized Assembly |
| ldr r3, [fp, #-20]  mov r0, r3 ;ldr into R0 instead  sub sp, fp, #4  ldmfd sp!, {fp, lr}  bx lr | ldr r0, [fp, #-20]  sub sp, fp, #4  ldmfd sp!, {fp, lr}  bx lr |

After applying the optimizations, the processor cycle count was reduced by 21 cycles. Aside from the Optimizations discussed, it was determined that the GCC compiler was effective in compiling the C code and did not create any necessary pipeline bubbles with unnecessary branch statements. Additionally, the compiler kept all local variables in registers, minimizing the processor’s memory accesses and further reducing pipeline bubbles.

It was noted that the assembly generated from the Lookup Table software solution featured many memory access instructions, indicating that the 16 KB lookup table could not be kept in cache. As a result of the memory access instructions the software solution was slower than expected and ranked second to the Reverse If Statement Solution when running on the ARM processor. The solutions were later tested on an Intel Core i7-9750H CPU @ 2.60GHz processor with a cache size of 14 MB. It was determined that the Lookup Table solution ran quickest beating out the Reverse If Statement Solution. From the results of the tests, one may conclude that the Lookup Table solution requires too much memory for an embedded system processor and an alternative solution such as the reverse if statement solution would be best.

A full breakdown showing the optimization made to the assembly as well as the stalling of the pipeline may be seen in **Appendix #** [6].

# 5.0 Performance

To evaluate performance of the software solutions discussed in section 3 of this report

The four software solutions, Operator Strength Reduction 𝜇 Law, Optimized 𝜇 Law with If Statements Reversed, Optimized 𝜇 Law with Switch Statement, and 𝜇 Law Lookup Table, have been run on the S3C2440A 32-Bit CMOS Microcontroller and compared with one another as well as the original, Unoptimized 𝜇 Law Software Solution, software solution [1]. The testing regime included evaluating each solution based on execution time required to compress and decompress a 1 MB Wav file. The audio file consisted of a voice speaking. This choice was made as 𝜇 Law audio compression has been designed to compress and decompress voice traffic. To ensure accurate results each solution was run nine times. The run times are recorded in Tables 12 to 16. The execution times were averaged and are presented in the Table 11 and Figure 22.

Table 11: Summary of the performance of the software solutions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Compression File Size | Decompression File Size | Compression Execution Time (Seconds) | Decompression Execution Time  (Seconds) |
| Unoptimized | 504 KB | 1MB | 0.842 | 0.834 |
| Optimized If Statement | 504 KB | 1MB | 0.7 | 0.8 |
| Optimized Reverse If Statement | 504 KB | 1MB | 0.583 | 0.764 |
| Lookup Table | 504 KB | 1MB | 0.61 | 0.78 |
| Optimized Switch Statement | 504 KB | 1MB | 1.04 | 0.9 |

Table 12: Optimized If Statement Execution Time

|  |  |
| --- | --- |
| Optimized If Statements | |
| Iteration | Execution Time |
| 1 | Audio Compression Time:0.670000 Seconds  Audio Decompression Time: 0.770000 Seconds |
| 2 | Audio Compression Time:0.680000 Seconds  Audio Decompression Time: 0.770000 Seconds |
| 3 | Audio Compression Time:0.680000 Seconds  Audio Decompression Time: 0.780000 Seconds |
| 4 | Audio Compression Time:0.690000 Seconds  Audio Decompression Time: 0.790000 Seconds |
| 5 | Audio Compression Time:0.690000 Seconds  Audio Decompression Time: 0.810000 Seconds |
| 6 | Audio Compression Time:0.680000 Seconds  Audio Decompression Time: 0.810000 Second |
| 7 | Audio Compression Time:0.720000 Seconds  Audio Decompression Time: 0.810000 Seconds |
| 8 | Audio Compression Time:0.730000 Seconds  Audio Decompression Time: 0.810000 Seconds |
| 9 | Audio Compression Time:0.740000 Seconds  Audio Decompression Time: 0.840000 Seconds |

Table 13: Optimized Reverse If Statement Execution Time

|  |  |
| --- | --- |
| Optimized Reversed If Statements | |
| Iteration | Execution Time |
| 1 | Audio Compression Time:0.620000 Seconds  Audio Decompression Time: 0.710000 Seconds |
| 2 | Audio Compression Time:0.640000 Seconds  Audio Decompression Time: 0.750000 Seconds |
| 3 | Audio Compression Time:0.650000 Seconds  Audio Decompression Time: 0.760000 Seconds |
| 4 | Audio Compression Time:0.660000 Seconds  Audio Decompression Time: 0.750000 Second |
| 5 | Audio Compression Time:0.650000 Seconds  Audio Decompression Time: 0.760000 Seconds |
| 6 | Audio Compression Time:0.680000 Seconds  Audio Decompression Time: 0.750000 Seconds |
| 7 | Audio Compression Time:0.670000 Seconds  Audio Decompression Time: 0.790000 Seconds |
| 8 | Audio Compression Time:0.680000 Seconds  Audio Decompression Time: 0.790000 Seconds |
| 9 | Audio Compression Time:0.700000 Seconds  Audio Decompression Time: 0.820000 Seconds |

Table 14: Look-Up Table Execution Time

|  |  |
| --- | --- |
| Optimized Look-Up Table | |
| Iteration | Execution Time |
| 1 | Audio Compression Time With Look Up Table:0.610000 Seconds  Audio Decompression Time: 0.770000 Seconds |
| 2 | Audio Compression Time With Look Up Table:0.600000 Seconds  Audio Decompression Time: 0.790000 Second |
| 3 | Audio Compression Time With Look Up Table:0.610000 Seconds  Audio Decompression Time: 0.780000 Seconds |
| 4 | Audio Compression Time With Look Up Table:0.630000 Seconds  Audio Decompression Time: 0.780000 Seconds |
| 5 | Audio Compression Time With Look Up Table:0.630000 Seconds  Audio Decompression Time: 0.790000 Seconds |
| 6 | Audio Compression Time With Look Up Table:0.640000 Seconds  Audio Decompression Time: 0.790000 Seconds |
| 7 | Audio Compression Time With Look Up Table:0.640000 Seconds  Audio Decompression Time: 0.790000 Seconds |
| 8 | Audio Compression Time With Look Up Table:0.670000 Seconds  Audio Decompression Time: 0.810000 Seconds |
| 9 | Audio Compression Time With Look Up Table:0.590000 Seconds  Audio Decompression Time: 0.750000 Seconds |

Table 15: Switch Statement Execution Time

|  |  |
| --- | --- |
| Optimized Switch Statement | |
| Iteration | Execution Time |
| 1 | Audio Compression Time:1.090000 Seconds  Audio Decompression Time: 1.320000 Seconds |
| 2 | Audio Compression Time:0.990000 Seconds  Audio Decompression Time: 0.880000 Seconds |
| 3 | Audio Compression Time:1.000000 Seconds  Audio Decompression Time: 0.860000 Seconds |
| 4 | Audio Compression Time:1.040000 Seconds  Audio Decompression Time: 0.890000 Seconds |
| 5 | Audio Compression Time:1.020000 Seconds  Audio Decompression Time: 0.850000 Seconds |
| 6 | Audio Compression Time:1.040000 Seconds  Audio Decompression Time: 0.900000 Seconds |
| 7 | Audio Compression Time:1.040000 Seconds  Audio Decompression Time: 0.880000 Seconds |
| 8 | Audio Compression Time:1.040000 Seconds  Audio Decompression Time: 0.920000 Seconds |
| 9 | Audio Compression Time:1.040000 Seconds  Audio Decompression Time: 0.920000 Seconds |

Table 16: Unoptimized Execution Time

|  |  |
| --- | --- |
| Original (unoptimized) | |
| Iteration | Execution Time |
| 1 | Audio Compression Time:0.920000 Seconds  Audio Decompression Time: 0.800000 Seconds |
| 2 | Audio Compression Time:0.810000 Seconds  Audio Decompression Time: 0.800000 Seconds |
| 3 | Audio Compression Time:0.940000 Seconds  Audio Decompression Time: 1.050000 Seconds |
| 4 | Audio Compression Time:0.810000 Seconds  Audio Decompression Time: 0.780000 Seconds |
| 5 | Audio Compression Time:0.830000 Seconds  Audio Decompression Time: 0.820000 Seconds |
| 6 | Audio Compression Time:0.810000 Seconds  Audio Decompression Time: 0.800000 Seconds |
| 7 | Audio Compression Time:0.820000 Seconds  Audio Decompression Time: 0.800000 Seconds |
| 8 | Audio Compression Time:0.820000 Seconds  Audio Decompression Time: 0.830000 Seconds |
| 9 | Audio Compression Time:0.820000 Seconds  Audio Decompression Time: 0.830000 Seconds |

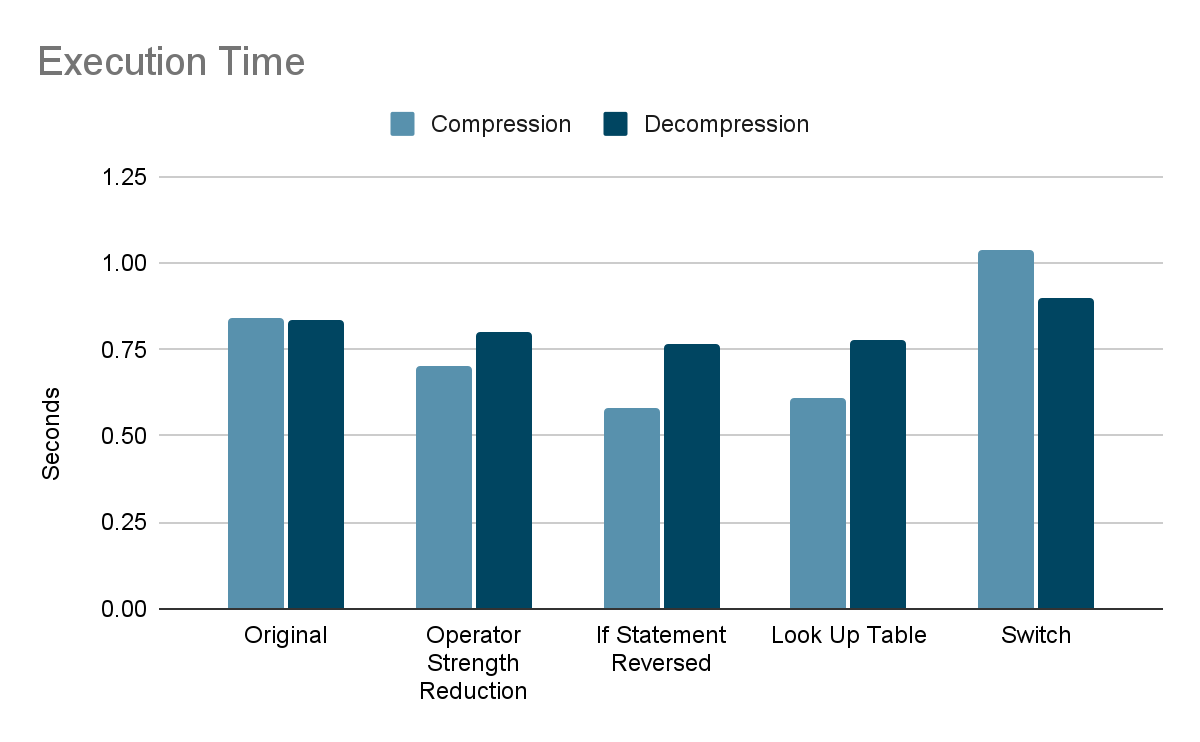


Figure 22: Execution Time of Software Solutions

Comparing the software solutions to one another, one may see that by using the fastest software solution, the If Statement Reversed, one may achieve a speed up of 31% over the original, Unoptimized solution. While this is a significant improvement, one may also wish to consider the speed of up a hardware solution. From the simulations discussed in, section 3.2, Hardware Solutions, results show an execution time of 90 micro-seconds. Comparing the hardware solution’s performance to the original, Unoptimized solution, one may observe a decrease in execution time by a factor of 9355. Comparing this value to the software solution’s decrease in execution time by a factor of 1.44, one may see that the hardware solution may provide a significant increase in performance. While the simulated results of the hardware solution are promising, it is a greater challenge to implement and would likely be financially expensive. Determining whether to choose the optimized software solution or the hardware solution would ultimately be dictated by whether the performance offered by the hardware solution is worth the financial cost over the less expensive software solution.

## 5.1 Audio Compression

As shown in Figure 22, the audio compression has a greater variance in execution times. This outcome is expected as all the software solutions feature different compression methods and the same decompression methods. From the figure one may note that the If Statement Reversed has the smallest execution time and has a file size like other solutions. Due to the solution’s supervisor performance, it is the primary recommendation for implementing 𝜇 Law audio compression and decompression.

Unexpectedly, the lookup table solution has the second smallest execution time oppose the first smallest. The cause of its slower execution time is likely due to cache misses. When comparing the lookup table solution to the if Statement Revered Solution using an Intel Core i7-9750H CPU @ 2.60GHz, one may observe that the lookup table runs fastest as the entirety of the 16 KB lookup table may be kept in cache.

## 5.2 Audio Decompression

The optimized software solutions, Operator Strength Reduction, If Statement Reversed, Lookup Table, and Switch Statement, feature the same decompression function. After reviewing the assembly for each software solution, it was found that the GCC compiler incorporated different memory management methods for the different software solutions, leading to different decompression execution times.

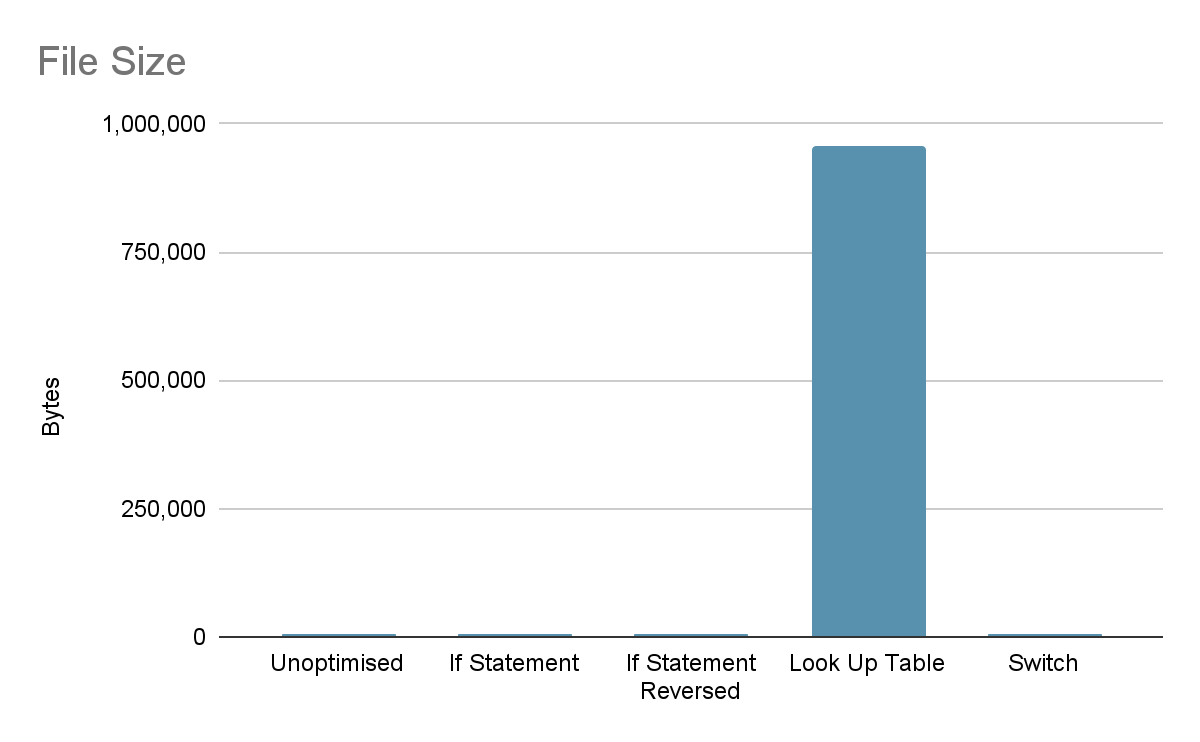


Figure 23: File Size of Software Solutions

Comparing the file size of the software solutions, one can observe the significantly larger file size of the lookup table. The lookup table file slide is large because the lookup table is composed of 16,383 entities. Due to the numerous entries the file size of the lookup table totaled 958,162 bytes. The original, Operator Strength Reduction, If Statement Reversed, and Switch software solutions has file sizes of 7,034, 7,252, 7,249, 7,417 bytes respectively. The reason they are all so similar is because they all contained a similar amount of branch statements. Additionally, they all used the same decompression method.

# 6.0 Results

When running out executable file on the lab machine, on “Matts-Voice.wav” the following compressed files was created:

A picture containing text, clock

Description automatically generated

Figure 24: Resulting Wave File Compression Illustration

Above, shows the created compressed.wav file showing that the solution was successful in reducing the size of the WAV file by half and then decompressed to achieve a file that was fairly close to the same size as the original file. The resulting audio was coherent; however the quality was diminished significantly. The difference in audio quality could be attributed to the data being lost during the original compression stage that was effectively compensated for during the decompression stage.

The program was able to compress the original audio file with a file size of 1,007,876 bytes to a compressed audio file with a file size 503,960 bytes, achieving a compression ratio of 2. Following the compression, the program was able to receive the compressed file and decompress it. Following decompression, the audio file was restored to its original file size. The decompressed audio file had significant noise compared to the original file, however, the audio is audible and coherent.

# 7.0 Conclusions

This project explored several software and compile time optimizations to improve the runtime performance of a 𝜇 Law audio compression and decompression algorithm implemented on an embedded S3C2440A 32-Bit CMOS Microcontroller. The following C programming optimization techniques that were made included: Operator Strength Reduction, If Statement Reversed, Lookup Table, and Switch Statement. Each of these techniques incorporated the same algorithm for decompression algorithm throughout the different code revisions.

After empirically testing these revisions on the embedded linux lab machine, the Reverse If Statement technique performed the best, yielding a 31% faster runtime than the original, unoptimized code. When testing each of these on the Intel Core i7-9750H CPU @ 2.60GHz onboard one of the team member’s personal Windows Machines, it was observe that the lookup table performed better, yielding the fastest execution time, due to the entirety of the 16 KB table may be kept in the processor’s cache. The smaller cache size of the embedded processor likely diminished any performance advantages of the lookup table where its enormous file size likely resulted in cache misses on the embedded processor. These results were consistent with our theoretical predictions.

When optimizing the assembly instructions through removing unnecessary memory store and load operations, as an attempt to further improve this operation we also declared all variables as the register keyword. Despite these optimizations, for this architecture, optimization techniques did not show any performance enhancements as the compiler continued to atomically create unnecessary stores and loads on new variables. In the future, this could be mitigated through assembly inclining.

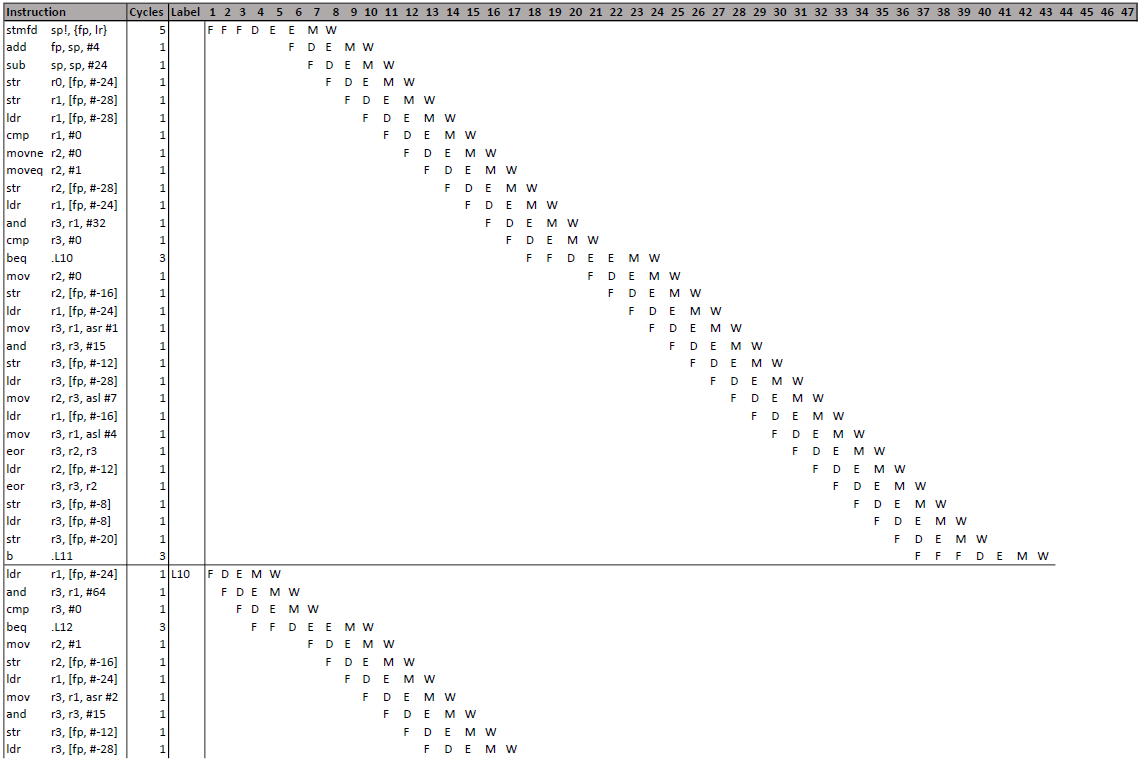
Ultimately, the fastest solution would be to implement this entire algorithm in hardware, as shown by the results that were simulated in this report. However, a designated hardware solution for this algorithm would be significantly more expensive financially due to the complexity, materials and overhead of implementing that entire solution onto an FPGA. Therefore, the optimal solution would be determined by the consumer as a trade-off between performance offered by the hardware solution compared to the overall financial benefits of the software solution.

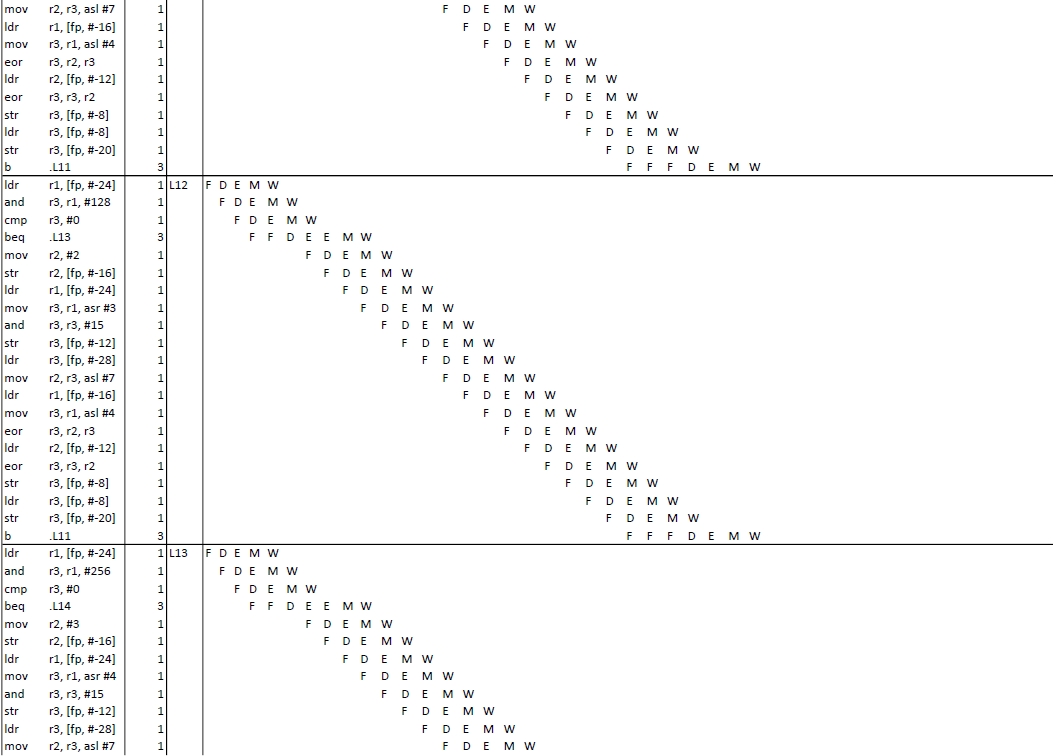
# 8.0 References

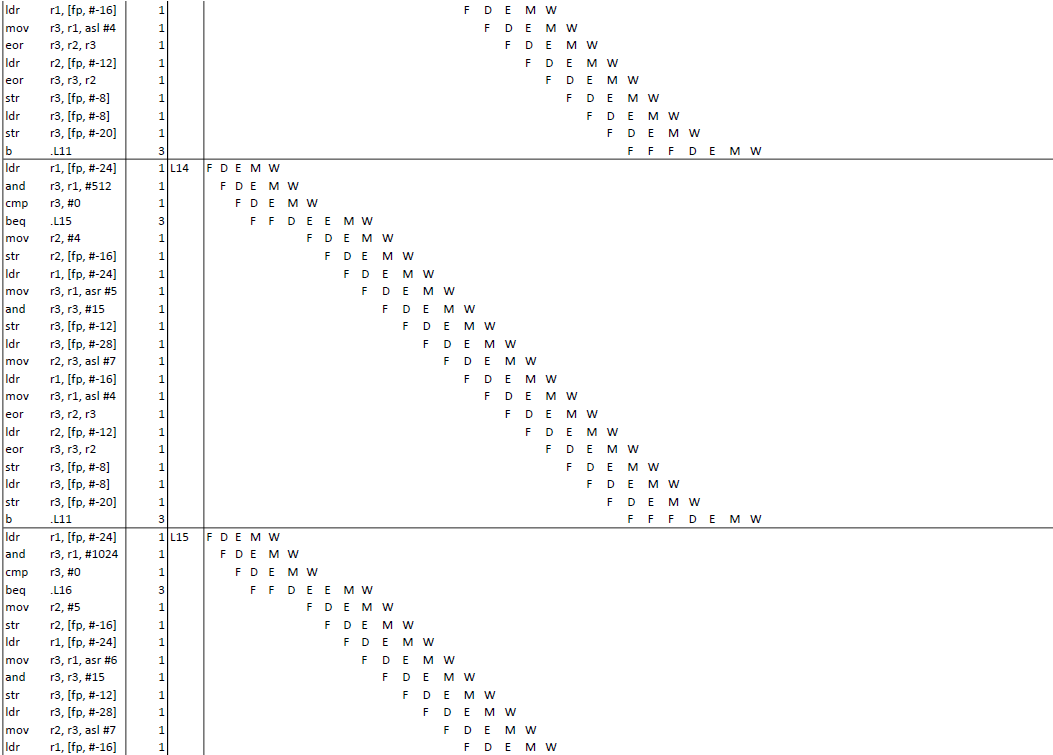
|  |  |
| --- | --- |
| [1] | M. Sima , “Seng440 Embedded Systems Lesson 104”. University of Victoria, 2021 |
| [2] | Jonathanhays, “Mu-Law and A-Law Compression Tutorial,” *Jonathan Hays*, 14-Nov-2018. [Online]. Available: https://jonathanhays.me/2018/11/14/mu-law-and-a-law-compression-tutorial/. [Accessed: 04-Aug-2021]. |
| [3] | *University of Victoria - sign in service*. [Online]. Available: https://bright.uvic.ca/d2l/le/content/128343/Home/ l/le/content/128343/viewContent/1101949/View [Accessed: 04-Aug-2021]. |
| [4] | “Companding: Logarithmic laws, implementation, and consequences - technical articles,” *All About Circuits*. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/companding-logarithmic-laws-implementation-and-consequences/. [Accessed: 04-Aug-2021]. |
| [5] | C. S. S. (craig@ccrma.stanford.edu), “WAVE PCM soundfile format,” *Microsoft WAVE soundfile format*. [Online]. Available: https://web.archive.org/web/20130510081513/https://ccrma.stanford.edu/courses/422/projects/WaveFormat/. [Accessed: 04-Aug-2021]. |
| [6] | *University of Victoria - sign in service*. [Online]. Available: https://bright.uvic.ca/d2l/le/content/128343/Home. [Accessed: 04-Aug-2021]. |

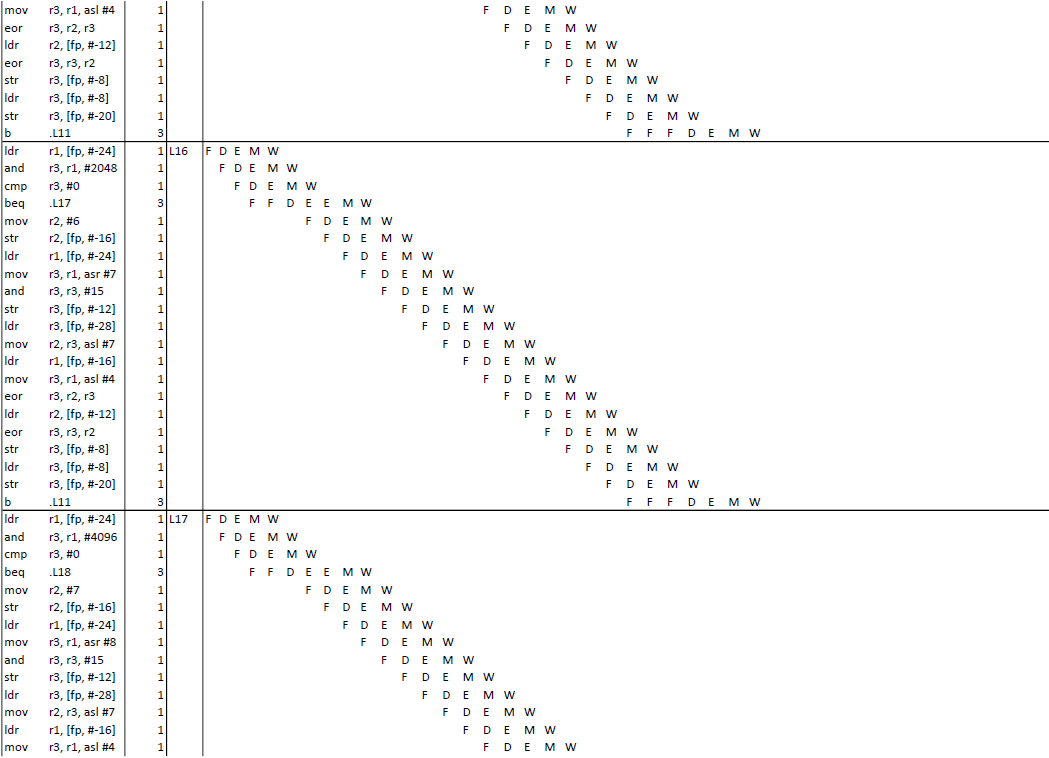
# 

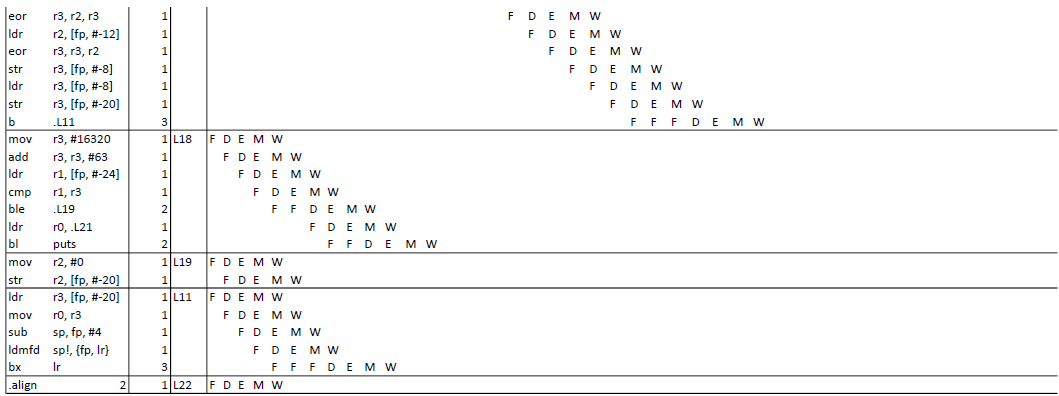
# Appendix ###: Original (unoptimized) Assembly Pipelining



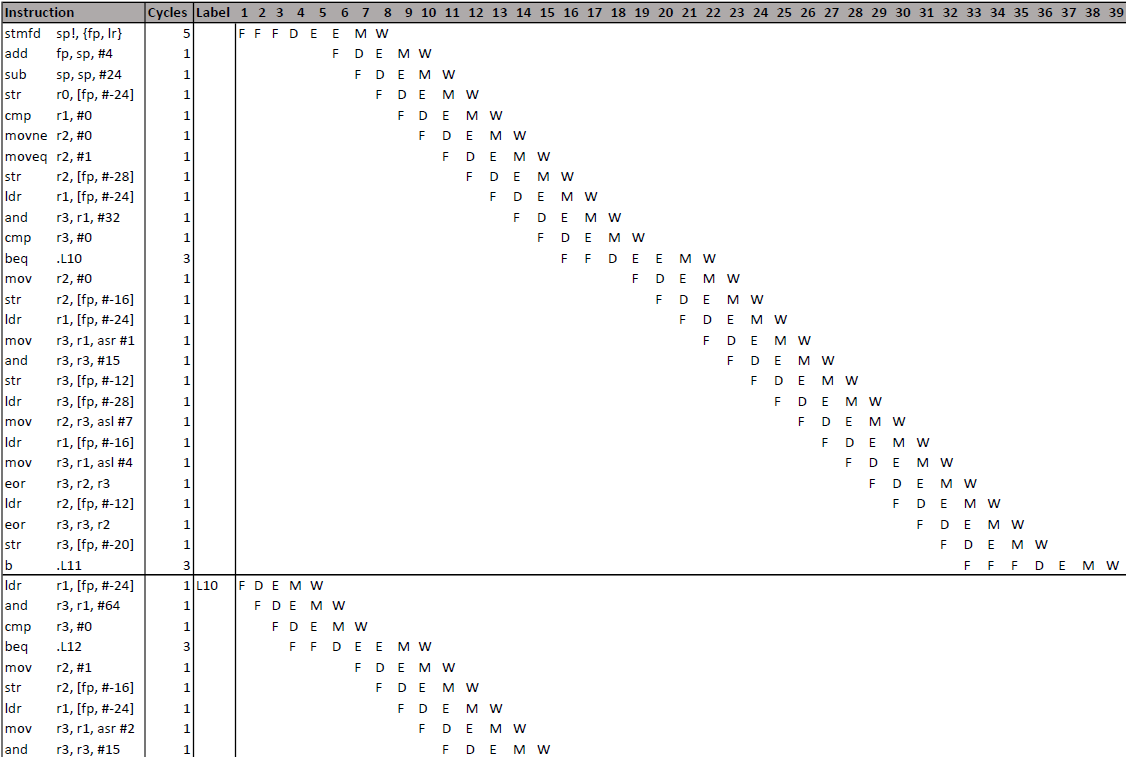


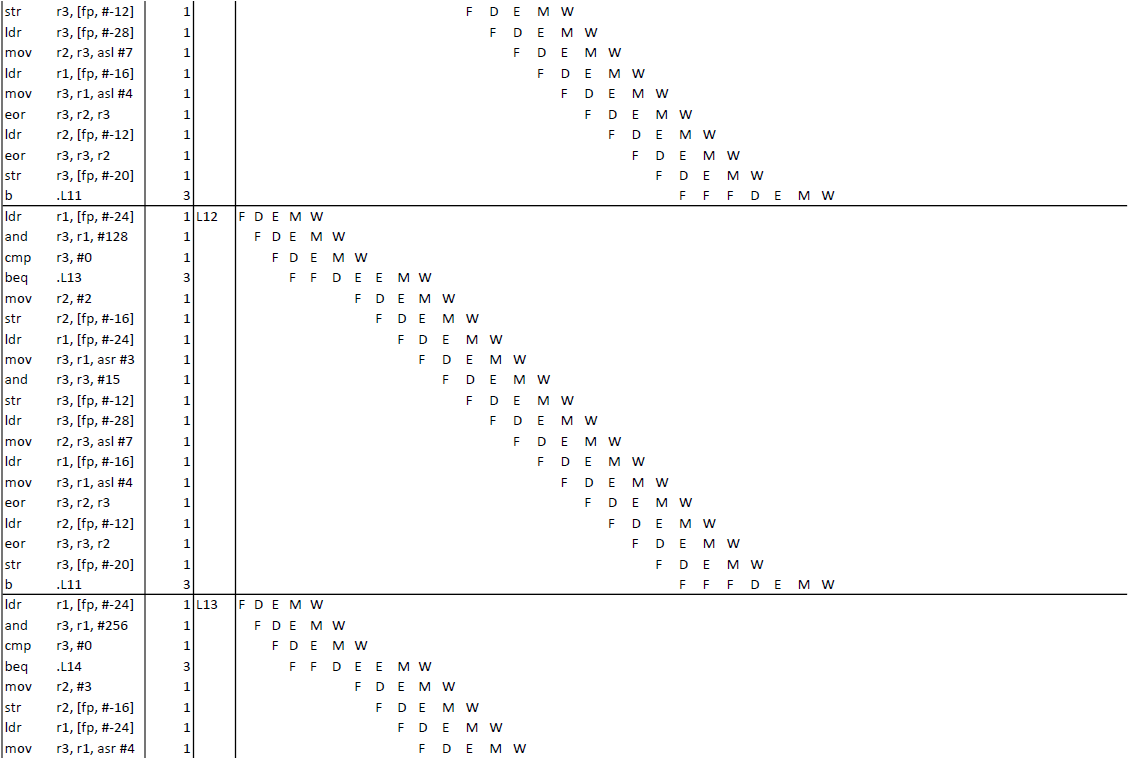


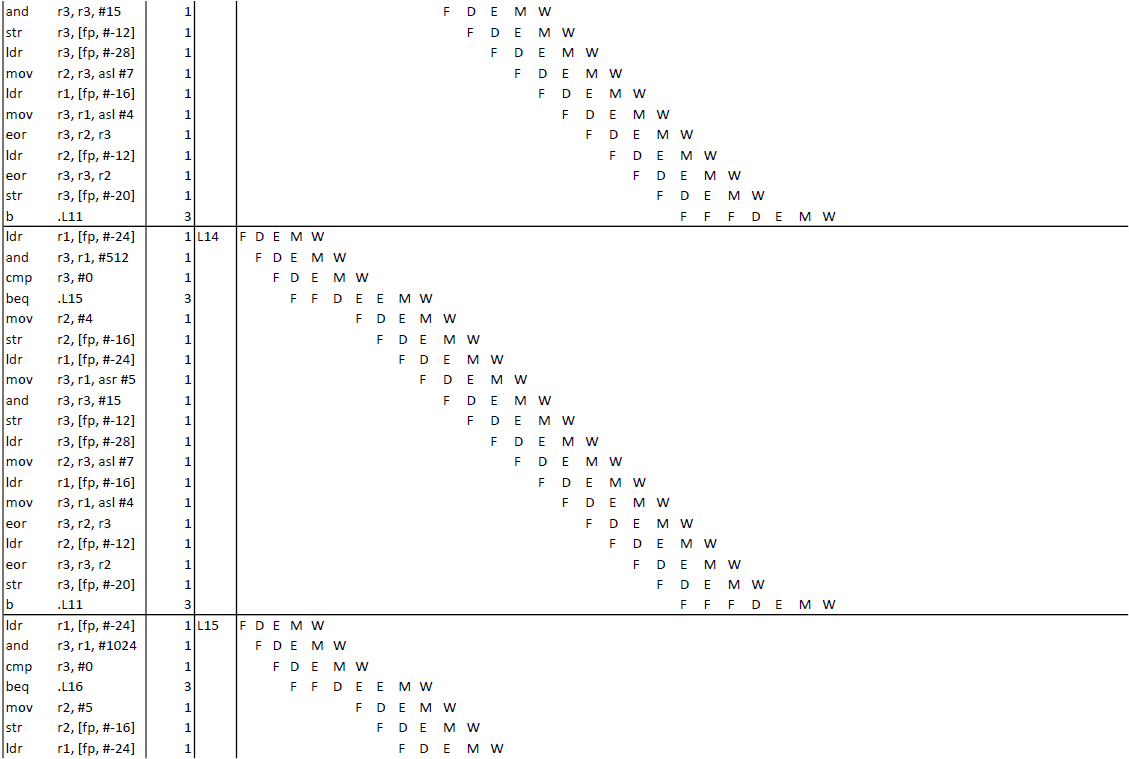


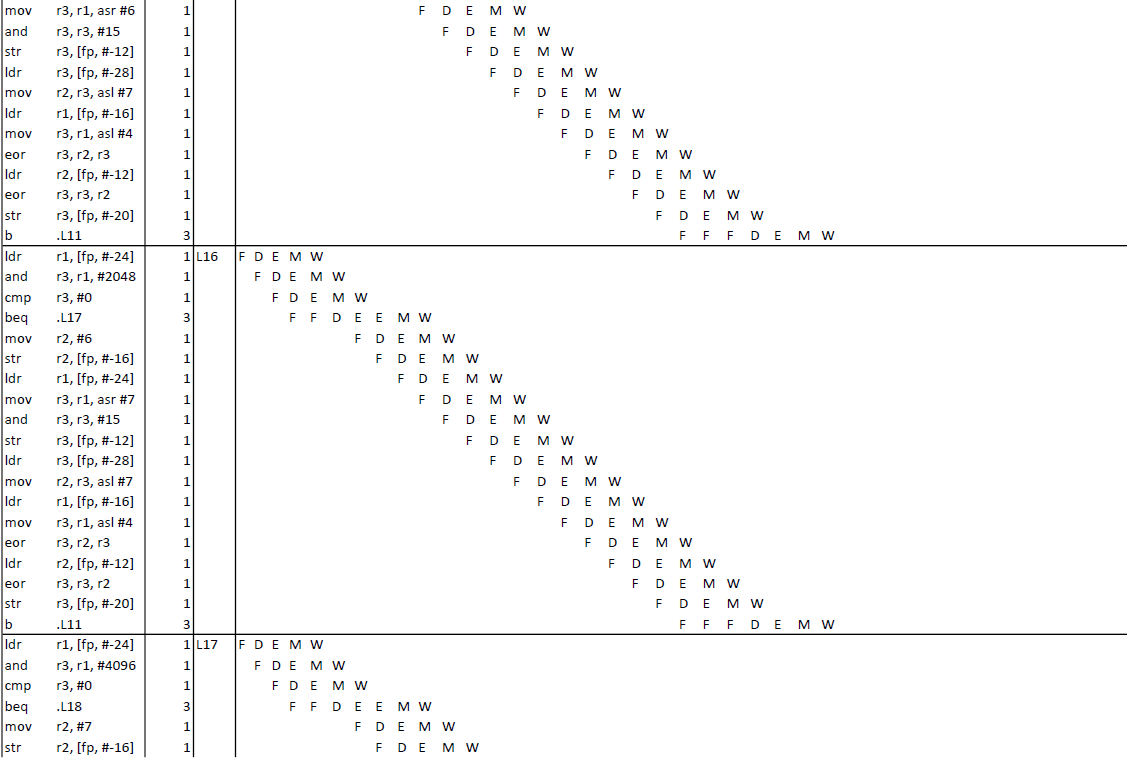


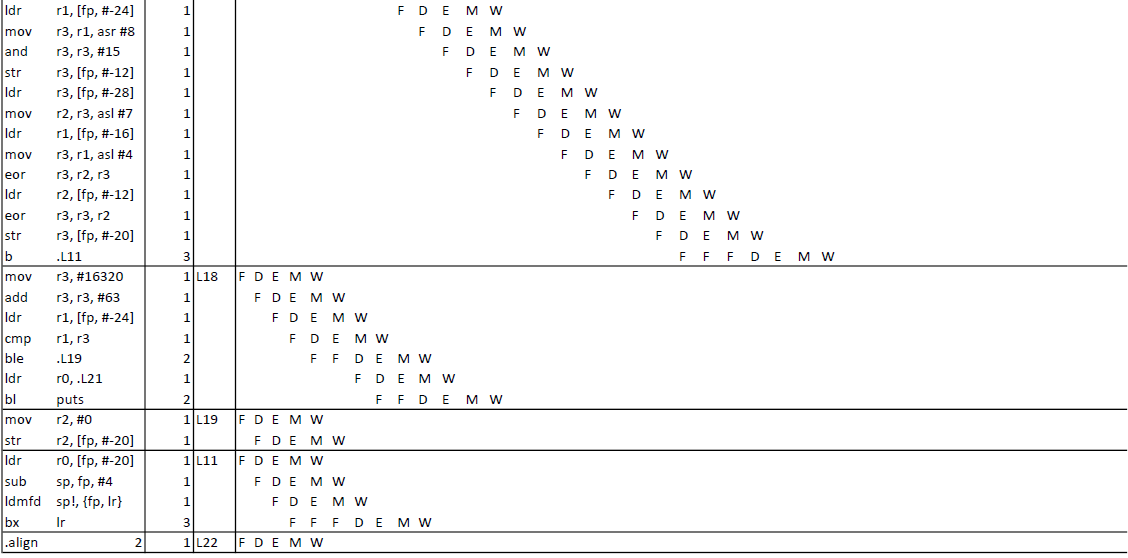
# Appendix ###: Optimized Assembly Pipelining











# Appendix E: VHDL Code for Audio Compression

--Audio Compression using 𝜇 Law

--Input 14 bits. First bit is a 0 and the following are data.

--The most significant four bits following the leading 1 are kept as 'step' and the remain bits are truncated.

--The size of the input value is stored as the data's 'chord'. this value is later used to decompress the data.

--The compressed bits consist of a leading 1 or 0 for sign (1 for positive) followed by three bits for the 'chord' followed by the step.

library IEEE;

use IEEE.STD\_LOGIC\_1164.all;

use IEEE.numeric\_std.all;

entity SourceF is

Port (

INPUT: IN std\_logic\_vector(13 downto 0);

OUTPUT: OUT std\_logic\_vector(7 downto 0)

);

end SourceF;

architecture Behavioral of SourceF is

signal input\_local: signed(13 downto 0);

signal output\_local : signed(6 downto 0);

signal chord : signed(2 downto 0);

signal step : signed(3 downto 0);

begin

process(INPUT) is

begin

input\_local <= signed(INPUT);

--Determine the signals chord and step

if(input\_local > 800) then

chord <= "111";

step <= resize((shift\_right(signed(input\_local), 8)), step ' length );

output\_local <= resize("1100000" OR (shift\_right(signed(input\_local), 8)), output\_local ' length );

elsif (input\_local > 400) then

chord <= "110";

step <= resize((shift\_right(signed(input\_local), 7)), step ' length );

output\_local <= resize("1100000" OR (shift\_right(signed(input\_local), 7)), output\_local ' length );

elsif(input\_local > 200) then

chord <= "101";

step <= resize((shift\_right(signed(input\_local), 6)), step ' length );

output\_local <= resize("1010000" OR (shift\_right(signed(input\_local), 7)), output\_local ' length );

elsif(input\_local > 100) then

chord <= "100";

step <= resize((shift\_right(signed(input\_local), 5)), step ' length );

output\_local <= resize("1000000" OR (shift\_right(signed(input\_local), 7)), output\_local ' length );

elsif(input\_local > 80) then

chord <= "011";

step <= resize((shift\_right(signed(input\_local), 4)), step ' length );

output\_local <= resize("0110000" OR (shift\_right(signed(input\_local), 7)), output\_local ' length );

elsif(input\_local > 40) then

chord <= "010";

step <= resize((shift\_right(signed(input\_local), 3)), step ' length );

output\_local <= resize("0100000" OR (shift\_right(signed(input\_local), 7)), output\_local ' length );

elsif(input\_local > 20) then

chord <= "001";

step <= resize((shift\_right(signed(input\_local), 2)), step ' length );

output\_local <= resize("0010000" OR (shift\_right(signed(input\_local), 7)), output\_local ' length );

elsif(input\_local > 10) then

chord <= "000";

step <= resize((shift\_right(signed(input\_local), 1)), step ' length );

output\_local <= resize(shift\_right(signed(input\_local), 7), output\_local ' length );

end if;

end process;

end Behavioral;

# Appendix F: Switch Statement VS If Statement Optimizations

Input: 1 | 1

Compressed: 0 |

Decompressed: -2147475456 |

0.002000 seconds elapsed to compress and decompress 1

Input: 2 | 10

Compressed: 0 |

Decompressed: -2147475456 |

0.002000 seconds elapsed to compress and decompress 2

Input: 5 | 101

Compressed: 0 |

Decompressed: -2147475456 |

0.003000 seconds elapsed to compress and decompress 5

Input: 12 | 1100

Compressed: 0 |

Decompressed: -2147475456 |

0.003000 seconds elapsed to compress and decompress 12

Input: 27 | 11011

Compressed: 0 |

Decompressed: -2147475456 |

0.003000 seconds elapsed to compress and decompress 27

Input: 58 | 111010

Compressed: 0 |

Decompressed: -2147475456 |

0.004000 seconds elapsed to compress and decompress 58

Input: 121 | 1111001

Compressed: 16 | 10000

Decompressed: 8225 | 10000000100001

0.006000 seconds elapsed to compress and decompress 121

Input: 248 | 11111000

Compressed: 32 | 100000

Decompressed: 8258 | 10000001000010

0.004000 seconds elapsed to compress and decompress 248

Input: 503 | 111110111

Compressed: 48 | 110000

Decompressed: 8324 | 10000010000100

0.003000 seconds elapsed to compress and decompress 503

Input: 1014 | 1111110110

Compressed: 64 | 1000000

Decompressed: 8456 | 10000100001000

0.003000 seconds elapsed to compress and decompress 1014

Input: 2037 | 11111110101

Compressed: 80 | 1010000

Decompressed: 8720 | 10001000010000

0.003000 seconds elapsed to compress and decompress 2037

Input: 4084 | 111111110100

Compressed: 0 |

Decompressed: -2147475456 |

0.002000 seconds elapsed to compress and decompress 4084

Input: 8179 | 1111111110011

Compressed: 96 | 1100000

Decompressed: 9248 | 10010000100000

0.002000 seconds elapsed to compress and decompress 8179

Input: 16370 | 11111111110010

Compressed: 112 | 1110000

Decompressed: 10304 | 10100001000000

0.003000 seconds elapsed to compress and decompress 16370

If Statement Performance

Input: 1 | 1

Compressed: 0 |

Decompressed: -2147475456 |

0.000000 seconds elapsed to compress and decompress 1

Input: 2 | 10

Compressed: 0 |

Decompressed: -2147475456 |

0.001000 seconds elapsed to compress and decompress 2

Input: 5 | 101

Compressed: 0 |

Decompressed: -2147475456 |

0.001000 seconds elapsed to compress and decompress 5

Input: 12 | 1100

Compressed: 0 |

Decompressed: -2147475456 |

0.001000 seconds elapsed to compress and decompress 12

Input: 27 | 11011

Compressed: 0 |

Decompressed: -2147475456 |

0.000000 seconds elapsed to compress and decompress 27

1101

Input: 58 | 111010

Compressed: 13 | 1101

Decompressed: -2147475443 |

0.001000 seconds elapsed to compress and decompress 58

11110

Input: 121 | 1111001

Compressed: 30 | 11110

Decompressed: 8253 | 10000000111101

0.001000 seconds elapsed to compress and decompress 121

101111

Input: 248 | 11111000

Compressed: 47 | 101111

Decompressed: 8318 | 10000001111110

0.002000 seconds elapsed to compress and decompress 248

111111

Input: 503 | 111110111

Compressed: 63 | 111111

Decompressed: 8444 | 10000011111100

0.002000 seconds elapsed to compress and decompress 503

1001111

Input: 1014 | 1111110110

Compressed: 79 | 1001111

Decompressed: 8696 | 10000111111000

0.002000 seconds elapsed to compress and decompress 1014

1011111

Input: 2037 | 11111110101

Compressed: 95 | 1011111

Decompressed: 9200 | 10001111110000

0.003000 seconds elapsed to compress and decompress 2037

1101111

Input: 4084 | 111111110100

Compressed: 111 | 1101111

Decompressed: 10208 | 10011111100000

0.002000 seconds elapsed to compress and decompress 4084

1111111

Input: 8179 | 1111111110011

Compressed: 127 | 1111111

Decompressed: 12224 | 10111111000000

0.002000 seconds elapsed to compress and decompress 8179

1111111

Input: 16370 | 11111111110010

Compressed: 127 | 1111111

Decompressed: 12224 | 10111111000000

0.002000 seconds elapsed to compress and decompress 16370

TESTING RESULTS

July 24

<========================= TESTING ============================>

Input: 3604 | 111000010100

Compressed: 236 | 11101100

Decompressed: 1824 | 11100100000

<========================= TESTING ============================>

Input: 253 | 11111101

Compressed: 175 | 10101111

Decompressed: 126 | 1111110

<========================= TESTING ============================>

Input: -181 |

Compressed: 38 | 100110

Decompressed: 8282 | 10000001011010

<========================= TESTING ============================>

Input: -1781 |

Compressed: 91 | 1011011

Decompressed: 9072 | 100011011100

# Appendix XXX ASM for Unoptimized Solution

|  |  |
| --- | --- |
| codeword\_compression: |  |
|  | @ Function supports interworking. |
|  | @ args = 0, pretend = 0, frame = 32 |
|  | @ frame\_needed = 1, uses\_anonymous\_args = 0 |
|  | stmfd sp!, {fp, lr} |
|  | add fp, sp, #4 |
|  | sub sp, sp, #32 |
|  | str r0, [fp, #-24] |
|  | str r1, [fp, #-28] |
|  | ldr r2, [fp, #-24] |
|  | mov r3, #8128 |
|  | add r3, r3, #63 |
|  | cmp r2, r3 |
|  | ble .L6 |
|  | ldr r3, [fp, #-24] |
|  | sub r3, r3, #8192 |
|  | str r3, [fp, #-24] |
|  | .L6: |
|  | ldr r3, [fp, #-28] |
|  | cmp r3, #0 |
|  | movne r3, #0 |
|  | moveq r3, #1 |
|  | str r3, [fp, #-28] |
|  | ldr r2, [fp, #-24] |
|  | mov r3, #16320 |
|  | add r3, r3, #63 |
|  | cmp r2, r3 |
|  | ble .L7 |
|  | ldr r0, .L18 |
|  | bl puts |
|  | mov r3, #0 |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L7: |
|  | ldr r2, [fp, #-24] |
|  | mov r3, #4080 |
|  | add r3, r3, #15 |
|  | cmp r2, r3 |
|  | ble .L9 |
|  | mov r3, #7 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #8 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L9: |
|  | ldr r2, [fp, #-24] |
|  | mov r3, #2032 |
|  | add r3, r3, #15 |
|  | cmp r2, r3 |
|  | ble .L10 |
|  | mov r3, #6 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #7 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L10: |
|  | ldr r2, [fp, #-24] |
|  | mov r3, #1020 |
|  | add r3, r3, #3 |
|  | cmp r2, r3 |
|  | ble .L11 |
|  | mov r3, #5 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #6 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L11: |
|  | ldr r2, [fp, #-24] |
|  | mov r3, #508 |
|  | add r3, r3, #3 |
|  | cmp r2, r3 |
|  | ble .L12 |
|  | mov r3, #4 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #5 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L12: |
|  | ldr r3, [fp, #-24] |
|  | cmp r3, #255 |
|  | ble .L13 |
|  | mov r3, #3 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #4 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L13: |
|  | ldr r3, [fp, #-24] |
|  | cmp r3, #127 |
|  | ble .L14 |
|  | mov r3, #2 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #3 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L14: |
|  | ldr r3, [fp, #-24] |
|  | cmp r3, #63 |
|  | ble .L15 |
|  | mov r3, #1 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #2 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L15: |
|  | ldr r3, [fp, #-24] |
|  | cmp r3, #31 |
|  | ble .L16 |
|  | mov r3, #0 |
|  | str r3, [fp, #-16] |
|  | ldr r3, [fp, #-24] |
|  | mov r3, r3, asr #1 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r3, [fp, #-16] |
|  | mov r3, r3, asl #4 |
|  | eor r2, r2, r3 |
|  | ldr r3, [fp, #-12] |
|  | eor r3, r2, r3 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-32] |
|  | b .L8 |
|  | .L16: |
|  | mov r3, #0 |
|  | str r3, [fp, #-32] |
|  | .L8: |
|  | ldr r3, [fp, #-32] |
|  | mov r0, r3 |
|  | sub sp, fp, #4 |
|  | ldmfd sp!, {fp, lr} |
|  | bx lr |
|  | .L19: |
|  | .align 2 |
|  | .L18: |
|  | .word .LC0 |
|  | .size codeword\_compression, .-codeword\_compression |
|  | .align 2 |
|  | .global codeword\_decompression |
|  | .type codeword\_decompression, %function |

# Appendix XXX ASM for Optimized Reverse If Statement Solution (best solution)

|  |  |
| --- | --- |
| codeword\_compression: |  |
|  | @ Function supports interworking. |
|  | @ args = 0, pretend = 0, frame = 24 |
|  | @ frame\_needed = 1, uses\_anonymous\_args = 0 |
|  | stmfd sp!, {fp, lr} |
|  | add fp, sp, #4 |
|  | sub sp, sp, #24 |
|  | str r0, [fp, #-24] |
|  | str r1, [fp, #-28] |
|  | ldr r1, [fp, #-28] |
|  | cmp r1, #0 |
|  | movne r2, #0 |
|  | moveq r2, #1 |
|  | str r2, [fp, #-28] |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #32 |
|  | cmp r3, #0 |
|  | beq .L10 |
|  | mov r2, #0 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #1 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L10: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #64 |
|  | cmp r3, #0 |
|  | beq .L12 |
|  | mov r2, #1 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #2 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L12: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #128 |
|  | cmp r3, #0 |
|  | beq .L13 |
|  | mov r2, #2 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #3 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L13: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #256 |
|  | cmp r3, #0 |
|  | beq .L14 |
|  | mov r2, #3 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #4 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L14: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #512 |
|  | cmp r3, #0 |
|  | beq .L15 |
|  | mov r2, #4 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #5 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L15: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #1024 |
|  | cmp r3, #0 |
|  | beq .L16 |
|  | mov r2, #5 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #6 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L16: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #2048 |
|  | cmp r3, #0 |
|  | beq .L17 |
|  | mov r2, #6 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #7 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L17: |
|  | ldr r1, [fp, #-24] |
|  | and r3, r1, #4096 |
|  | cmp r3, #0 |
|  | beq .L18 |
|  | mov r2, #7 |
|  | str r2, [fp, #-16] |
|  | ldr r1, [fp, #-24] |
|  | mov r3, r1, asr #8 |
|  | and r3, r3, #15 |
|  | str r3, [fp, #-12] |
|  | ldr r3, [fp, #-28] |
|  | mov r2, r3, asl #7 |
|  | ldr r1, [fp, #-16] |
|  | mov r3, r1, asl #4 |
|  | eor r3, r2, r3 |
|  | ldr r2, [fp, #-12] |
|  | eor r3, r3, r2 |
|  | str r3, [fp, #-8] |
|  | ldr r3, [fp, #-8] |
|  | str r3, [fp, #-20] |
|  | b .L11 |
|  | .L18: |
|  | mov r3, #16320 |
|  | add r3, r3, #63 |
|  | ldr r1, [fp, #-24] |
|  | cmp r1, r3 |
|  | ble .L19 |
|  | ldr r0, .L21 |
|  | bl puts |
|  | .L19: |
|  | mov r2, #0 |
|  | str r2, [fp, #-20] |
|  | .L11: |
|  | ldr r3, [fp, #-20] |
|  | mov r0, r3 |
|  | sub sp, fp, #4 |
|  | ldmfd sp!, {fp, lr} |
|  | bx lr |
|  | .L22: |
|  | .align 2 |
|  | .L21: |
|  | .word .LC1 |
|  | .size codeword\_compression, .-codeword\_compression |
|  | .align 2 |
|  | .global codeword\_decompression |
|  | .type codeword\_decompression, %function |