Fast-Integer Optimization for WMA Compatible Decoder on Embedded System without FPU

doi:10.6180/jase.2008.11.4.07

淡江理工學刊, 11(4), 2008

Journal of Applied Science and Engineering, 11(4), 2008

作者/Author： Lain-Jinn Hwang;Chien-Chou Shih;Wei-Chen;I-Ting Kuo

頁數/Page： 375-386

出版日期/Publication Date：2008/12

Fast-Integer Optimization for WMA Compatible Decoder on Embedded System without FPU

Lain-Jinn Hwang[1], Chien-Chou Shih[2]*, Wei-Chen[3] and I-Ting Kuo[1]

[1]*Department of Computer Science and Information Engineering, Tamkang University, Tamshui, Taiwan 251, R.O.C.*
[2]*Department of Information and Communication, Tamkang University, Tamshui, Taiwan 251, R.O.C.*
[3]*Department of Computer Science and Application, Asia University, Taichung, Taiwan 413, R.O.C.*

Abstract

This paper presents a general software optimization technique which enables an embedded system to play Windows Media Audio (WMA) fluently without the support of floating point unit (FPU). We employ fixed-point arithmetic operations, instead of floating-point, to optimize the computational overhead during the audio decoding process. Thus the proposed performance improvements by programming in C language are useful for the implementation of the real-time WMA compatible decoder on ARM920T based embedded system. This work achieved performance increase by reducing the CPU usage rate from 100% to 45% with great precision on the average around 1-bit error.

Key Words: Ebedded Sstem, Foating Pint, Fxed-point, WMA

1. Introduction

1.1 Motivations

The capability of providing real-time multimedia player over the Internet is an important future application for embedded system. However, the main challenge in such application is the performance of processing complex computations, especially in decoding processes. Several researches devoted to the development of embedded applications to enable the multimedia functionalities, such as MP3 audio or MPEG-2 video decoder [1−5] with low cost and lightweight devices. Lee et al. have developed the architecture for MP3 decoding system based on a processor employing a dual-core (DSP and RISC) architecture [6]. However, the computation performances in decoding and encoding are usually limited in the architecture of embedded system. For instance, many familiar embedded systems were designed without being equipped with the DSP core. That is, even if the MP3 [7] and Ogg [8] integer-only decoder (e.g. **MAD** [9] or **Ogg-tremor** [9]) was found, we still could not play WMA file on embedded systems such as S3C2440, XSCALE-PXA255 or XSCALE-PXA270, etc. On the other hand, Windows Media Audio (WMA) is a closed digital audio format developed by Microsoft [10]. It can be reverse-engineered and/or re-implemented by **FFmpeg** software today. However, to play WMA audio fluently on the embedded system without a floating-point processor, the floating-point calculations may be simulated by software. But such simulator cannot calculate massive floating-point operations at the same time. Based on the above consideration, the goal of this paper is to propose the embedded software optimization technique based on fast integer methodology, so as to improve the performance of WMA compatible decoder on the LINUX-based embedded system without FPU processor.

1.2 Related Work

MP3 and Ogg are two of the most popular audio formats, which are developed as integer-only decoders.

*Corresponding author. E-mail: ccs@mail.tku.edu.tw

MAD is also a MP3 audio decoder that fully supports fixed-point computation and is developed by Underbit Technologies Inc. [11]. Tremor is a fixed-point implementation of the Ogg Vorbis decoder developed by Xiph open source community [9]. Thus it can be seen that the integer-only decoders are widely used and made it easy to play MP3 and Ogg on the portable player devices. However, when considering the performance of real-time playing MP3 audio files on a non-DSP embedded system, it needs optimization methods to deal with complex computations. Yao et al. [3] have proposed general hardware & software co-optimization techniques for embedded systems based on RISC32 processor to optimize MP3 decoder. However, the decoding algorithm of WMA audio is relatively more complex than MP3. Therefore, it is more challenging to implement WMA decoder efficiently on a non-DSP embedded system.

The remainder of this paper is organized as follows. Section 2 describes the background of WMA decoding process and section 3 discusses the advantages of fixed-point. The proposed method of fixed-point optimization will be presented in section 4, and section 5 is the experimental results. Finally, the conclusions and future research are discussed in section 6.

2. Background of WMA Audio Decoder

In this section, how the **FFmpeg** WMA decoder works is discussed [12,13]. Since Microsoft did not release any details of the WMA decoder specifications, we have to infer the encoding algorithms according to the fundamental of audio data compression. Fundamentally, WMA is a transform coder based on modified discrete cosine transform (MDCT), somewhat similar to AAC and Vorbis. The bit stream of WMA is composed of superframes, each containing 1 or more frames of 2048 samples. If the bit reservoir is not used, a frame is equal to a super-frame. Each frame contains a number of blocks, which are 64, 128, 256, 512, 1024, or 2048 samples long after being transformed into the frequency domain via the MDCT [14]. In the frequency domain, masking for the transformed samples is determined, and then used to re-quantize the samples. Finally, the samples are Huffman coded. Stereo information is typically mid/side coded. At low bit rates, line spectral pairs (typically less than 17 kbit/s) and a form of noise coding (typically less than 33 kbit/s) can also be used to improve its quality. Thus, the audio compression has to pass the data transformation such as MDCT, for converting sound data from the time domain to the frequency domain, and then in an effort to avoid pre-echo artifacts.

Hence, after looking at the source code and studying how the **FFmpeg** WMA decoder is implemented, we can summarize the following WMA decoding phases.

1. Initialization for WMA Decoding

At the beginning, the sine and cosine transform table are initialized. The related equations are as follow:

$$\sin_i = -\sin\left(2\pi * \frac{i+\frac{1}{8}}{n}\right)_i \tag{1}$$

$$\cos_i = -\cos\left(2\pi * \frac{i+\frac{1}{8}}{n}\right)_i \tag{2}$$

$$\text{where } i = \begin{cases} 0,\ \ldots,\ 2^k,\ \text{and } k = 4,\ 5,\ \ldots,\ 10 \\ n,\ \ldots,\ 2^m,\ \text{and } m = 6,\ 7,\ \ldots,\ 12 \end{cases}$$

The next step is to initialize the exponent table and the bit reverse table used on FFT [15]. Besides the inverse modified discrete cosine transform (IMDCT) initialization, there are still some initializations such as windowing initialization, noise table initialization and so on. Figure 1 shows the initialization processes.

The equation (3) and (4) show the exponent tables:

$$\exp_{i,real} = \cos\left(2\pi * \frac{i}{n}\right)_i \tag{3}$$

$$\exp_{i,imaginary} = \sin\left(2\pi * \frac{i}{n}\right)_i \tag{4}$$

where $i = 0, 1, \ldots, 2^m$, and $m = 3, 4, \ldots, 9$;
where $n = 2^k$, and $k = 4, 5, \ldots, 10$

The data referenced from bit reverse table is already an integer number, therefore there is no more conversion needed. The window function as (5) will multiply the output of IMDCT and then obtains the frameout.

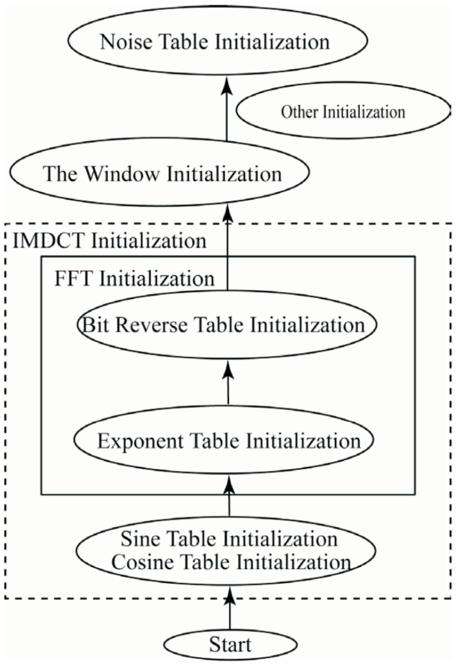

Noise Table Initialization
Other Initialization
The Window Initialization
IMDCT Initialization
FFT Initialization
Bit Reverse Table Initialization
Exponent Table Initialization
Sine Table Initialization
Cosine Table Initialization
Start

Figure 1. Initialization flow chart for WMA decoding.

$$windows_k = \sin\left((k+0.5)*\left(\frac{\pi}{2*n}\right)\right) \tag{5}$$

where $n = 2^{j-i}$, and $i = 1, 2, \ldots, 5$, and $j = 9, 10, 11$, and $k = 0, 1, \ldots, 2^m$, and $m = 5, 6, \ldots, 11$

Next, there is a random number to initialize the noise table with the initial value of *seed* being 1. As shown in (6), the noise table initialization is noise allocation because of low-bit rate or when the scale factor band is too large. The details are omitted here.

$$\text{noise table}_i = seed * norm \tag{6}$$

where $i = 0, 1, \ldots, 8191$, and
$seed = seed * 314159 + 1$, and

$$norm = \left(\frac{1}{1 << 31}\right) * \sqrt{3} * f$$

where $f = 0.02$ or 0.04

2. Inverse MDCT (IMDCT) Calculation

IMDCT is the inverse modified discrete cosine transforms [16]. Briefly, it subdivides the data sample into the finer band, and converts the time domain into frequency domain in order to reduce data. The difference between IDCT and IMDCT is that IMDCT includes the operations of window normalization and anti-aliasing. IMDCT function can be divided into four steps: pre-rotation, fast Fourier transform (FFT) [17], Post-rotation and re-order (Figure 2). The Pre-rotation which is built initially by using the sine, cosine and bit reverse table structure. In this part, there is a reverse index table which maps to the number conversely, and those complex numbers multiply each other to obtain a new number matrix, and then as the input of FFT.

Figure 3 shows the FFT pre-rotation, and the pre-rotation formula is as follows:

$$\begin{aligned}
&rptr\text{: rear pointer, } fptr\text{: front pointer,}\\
&stab\text{: sine table, } ctab\text{: cosine table,}\\
&Z_{i,\,real} = rptr * ctab_j - fptr * stab_j,\\
&Z_{i,\,imaginary} = rptr * stab_j + fptr * ctab_j,\\
&\text{where } i = \text{bit reverse table}_j,\\
&\text{and } j = 2^k, \text{ where } k = 2, 3, \ldots, 8.
\end{aligned} \tag{7}$$

The FFT calculation will convert wave data to a series of coefficient, which compresses with Huffman coding at the end. Post rotation takes the number that is ac-

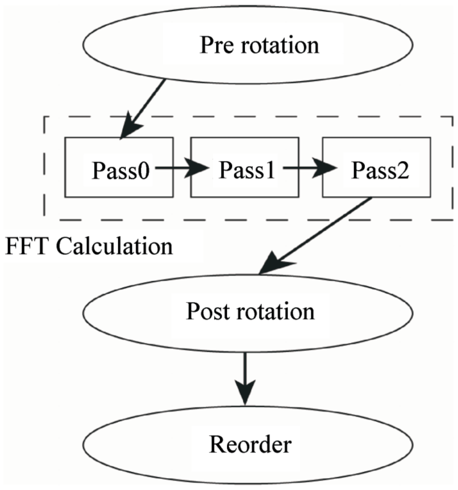


Figure 2. IMDCT calculation.

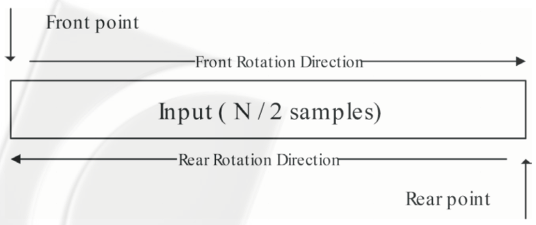


Figure 3. FFT pre-rotation

quired from FFT output to multiply the cosine or sine table. Finally, data are built by order.

As shown in Figure 2, a FFT calculation is divided into three passes. The first pass (pass 0) is implemented with butterfly operation [18], and which takes two values that are produced by pre-rotation to calculate. The front pointer *p* maps the corresponding address to FFT complex number structure. The structure is shown in Figure 4.

The second pass is similar to the first pass, as shown in Figure 5, but it takes two turns of butterfly operation in the first pass as a unit. The data is selected based on its index value. That is, when the index value is odd, the computing coefficient will be inverted.

The final pass is a complex computation because of the pointer structure. There is a structure to save the temporary complex data by pointers *p* and *q*. In the beginning, pointers *p* and *q* perform the butterfly operation, and both pointers move to the next address afterward as shown in Figure 6. After the butterfly operation, the pointer *q* multiplies the exponent table number built before, and we obtain the data alias to the temporary address. The foregoing steps will repeat persistently until all blocks have been computed.

As previous step, the data acquired from FFT multiplying the sine and cosine tables, which will be recorded to a matrix structure at the end of IMDCT algorithm. Besides, in the multiplication instruction cycle time in 32-bit RISC processor, the integer multiplication cycle is four per instruction [19], and the floating-point multiplication cycle is twenty times the clock cycle of the integer multiplication. Hence, to optimize the decoding performance by fix-point technology is obviously the best solution.

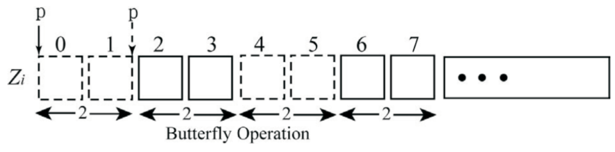


Figure 4. FFT Pass 0.

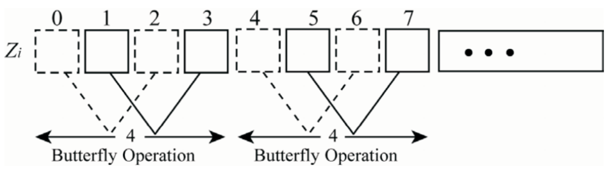


Figure 5. FFT Pass 1.

3. Fixed-Point Arithmetic

3.1 Fixed-Point Number Representation

A fixed-point number representation is a real data type for a number of fixed digits before and after the decimal point [20]. Fixed-point number representation is in contrast to the more flexible floating-point number representation. Most low-cost embedded processors do not have a floating-point unit. Therefore, the fixed-point numbers are useful to represent fractional values in native two's complement format if the executing processor has no floating-point unit (FPU) or if fixed-point is used to improved performance as well as accuracy. A fixed-point number may be represented as *M.F*, where *M* represents the magnitude, and *F* represents the fractional part. Each fractional bit represents an inverse power of 2. Thus the first fractional bit is ½, the second is ¼ and so on. For signed fixed-point numbers in two's complement format, the upper bound is given by $2^{m-1} - 2^{-f}$, and the lower bound is given by -2^{m-1}, where *m* and *f* are the number of bits in *M* and *F* respectively. For unsigned values, the range is 0 to $2^{m} - 2^{-f}$.

3.2 Advantages of Fixed-Point Implementation

The codec for the Ogg Vorbis [8] uses fixed-point arithmetic because many audio decoding hardware devices do not have an FPU, and audio decoding requires a sufficient amount of performance, that a software implementation of floating-point on low-speed devices can not

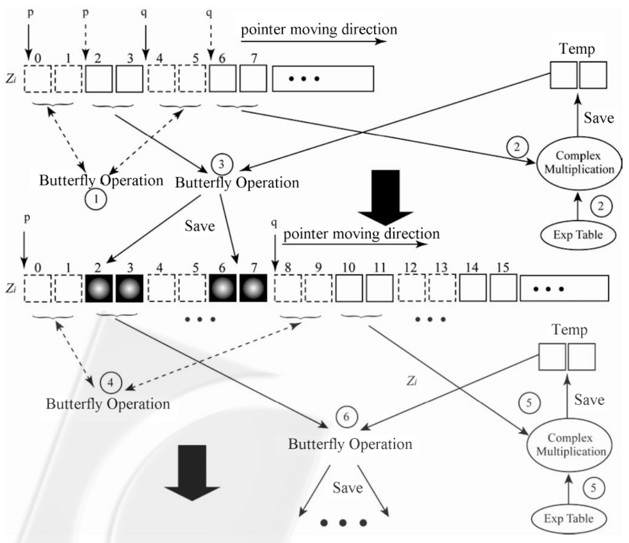


Figure 6. FFT Pass 2.

produce output in real time. Moreover, since the audio data is quantized in 16-bit, the truncation problem does not be encountered. With an integer size, the audio data can be represented completely. Therefore, the fixed-point implementation can meet the requirements for bit-accuracy and speed optimization.

According to the discussion in previous sections and the experiment in [3], IMDCT and sub-band synthesis routines are characterized by the multiply-accumulation operations with cosine coefficients during WMA/MP3 audio decoding. Since these routines require heavy computational loads, it is significant to perform modification by using fast integer optimization.

4. Fast Integer Optimization

On the single core embedded system such as SBC-2410x, there is no FPU to execute floating-point computation, and the calculations are usually simulated with integer operations so that the calculation speed is slow. Thus, playing WMA file on SBC-2410x will overload with floating point operations. In order to achieve the goal of this paper, we have proposed a general fast fixed-point optimum procedure, as shown in Figure 7, by programming in C language. Although C language is still the most popular and flexible for the development of digital signal processing algorithms as stated in [21], C does not support fixed-point format. The first phase of the proposed procedure is to define the conversions from floating-point functions into fix-pointed versions (i.e. FF_CONV, FF_REV and FF_ROUND in Figure 7) as well as the mathematical operations for fixed-point data type [21]. As a result, the fixed-point arithmetic operations are applicable to IMDCT calculation. On the other hand, we have modified the floating-point function such as power, square and root for optimizing the accuracy. Figure 8 shows the defined fixed-point data format, and the represented range is from 0x80000000 (-65536.000000) to 0x7FFFFFFF (65535.999969).

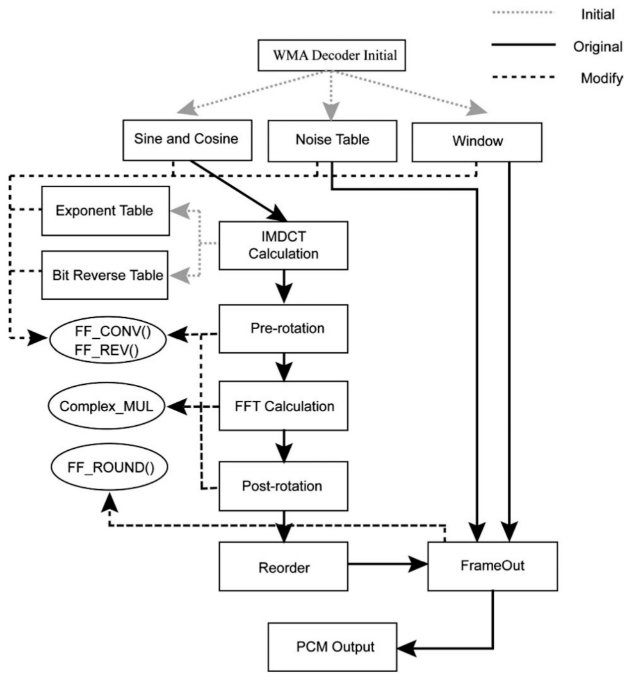


Figure 7. Overall procedure of fast integer decoding optimization

4.1 Conversion of Floating-Point Function

As discussed in the previous section (Figure 1), there are four tables when initializing IMDCT. The modified conversion functions are applied to the cosine and sine table. Likewise, the functions involved with complex multiplication are also revised by the fixed-point conversions. Furthermore, since the FFT algorithm is found in butterfly operation, which passes will modify the multiplication functions. Further refinement in other functions will be made to improve the redundant computation. The conversion function between integer and fixed-point is defined as formula (8) to improve the conversion time and to enhance its accuracy. The formula (9) is a reversion function.

$$\begin{aligned}&\text{(fixed) FF\_CONV}(X_{float})\\&\quad= \text{(signed integer)}(\,X_{float} * (1 << 15) + 0.5)\end{aligned} \tag{8}$$

$$\text{(float) FF\_REV(Yfixed)} = \text{(float)}\left(\frac{\text{(float)}Y_{fixed}}{1 << 15}\right) \tag{9}$$

Though the floating-point have been changed into fixed-point format, the data type of audio output must be integer. Hence, a shift function, as shown in formula (10), is utilized to round up the output for precision.

$$\begin{aligned}&\text{(integer)FF\_ROUND}(X_{fixed}) = X_{integer}\\&\quad= (X_{fixed} + (1 << (15 - 1))) >> 15\end{aligned} \tag{10}$$

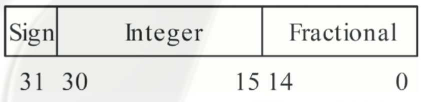


Figure 8. Fixed-point data format

4.2 Fixed-Point Mathematical Operations

The operations associated with the fixed-point arithmetic, such as "+," "-," "*," and "/" are also defined in this section. Four fundamental operations were implemented instead of the original operation and the complex multiplication, as shown in (11) and (12). The addition operation (13) and subtraction operation (14) are as usual. The multiplication (15) truncated is redefined in Figure 9, and the division operation, as shown in (16), can be done by multiplying the reciprocal with mathematic theory. Assume that *X* is divided by *Y*, we shift the fractional part bits twice with an integer number one as numerator, and divide by divisor *Y* to get the reciprocal. Then, we multiply *X* by the reciprocal *Y*.

In IMDCT implementation, there are many complex multiplications. The complex number includes real number and imaginary number, and it is represented by two floating-point numbers. The complex multiplication equations are as formula (11), and (12).

$$Z_{real} = X_{real} * Y_{real} - X_{imagin} * Y_{imagin} \tag{11}$$

$$Z_{imagin} = X_{real} * Y_{imagin} + X_{imagin} * Y_{real} \tag{12}$$

Four basic mathematical operations are defined as the following equations (13) to (16).

$$\text{FF\_ADD}(X, Y) = \text{sum}(X, Y) = X + Y \tag{13}$$

$$\text{FF\_SUB}(X, Y) = \text{difference}(X, Y) = X - Y \tag{14}$$

$$\begin{aligned}&\text{FF\_MUL\_TRUNC}(X, Y) = \text{product}(X, Y)_{64bits}\\ &\quad = (X_{32bits} * Y_{32bits}) >> 15\end{aligned} \tag{15}$$

$$\begin{aligned}&\text{FF\_DIV}(X, Y) = \text{quotient}(X, Y)\\ &\quad = \text{product}(X, \text{reciprocal}(Y))\end{aligned} \tag{16}$$

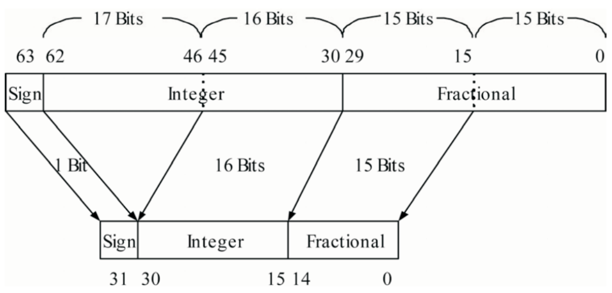


Figure 9. Fixed-point Multiplication Truncation.

where $\text{reciprocal}(Y)_{fixed\_64bits} = \dfrac{1 << (15 * 2)}{Y_{fixed\_32bits}}$

To reduce the CPU consumption and avoid the degradation of the sound quality, both the computational complexity and the data precision should be optimized.

4.3 Computational Complexity Optimization

1. Integer to fixed-point

Since the operation functions are fixed-point data format, the integer number is unable to multiply by fixed-point number directly. That is, when the bit reverse table is saved by integer, before referring to the value on the table, the following equation (17) is used to convert integer to fixed-point.

$$\text{(fixed) FF\_CONVI}(X_{integer}) = \text{X}_{\text{integer}} << 15 \tag{17}$$

2. Fixed-point multiplication

As shown in Figure 10, the implementation of the fixed-point multiplication function is programmed in ARM assembly language with instructions "*smull*", "*movs*" and "*adc*" to shorten a 64-bit register to a 32-bit register.

3. Complex multiplication

The complex multiplication is another problem. Except for fixed-point multiplication, the complex multiplication also can be optimized in assembly language. Different from fixed-point multiplication, there are additional instructions, "*rsb*" and "*smlal*", for inversion and multiplication.

4. The power and square-root functions

The parameter of power function can be simplified according to different case. For example, the power function such as (18) [22] can be converted into fixed-point directly.

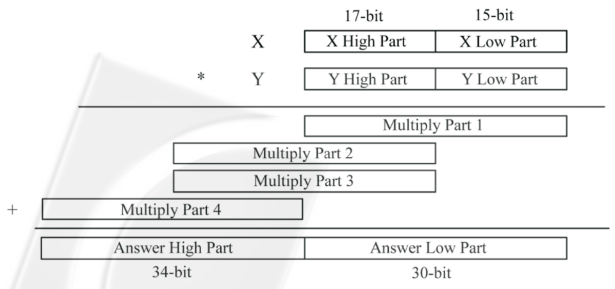


Figure 10. Fixed-point Multiplication.

$$\text{pow\_table}_{i.float} = \text{POW}(10, i * 0.05) \quad (18)$$

where $i = 0, 1, \ldots, 127$.

Since the range of i is perceivable, the problem of superfluous computation can be avoided by building a lookup table. The maximum value produced by this function is:

$$max = 10^{i*0.05} = 10^{6.35} = 2238721.138568$$

This value is over the range of a fixed-point as we have defined in Figure 8. Futhermore, there is a floating-point X involved in the power table and square-root function. To avoid the overflow problem, a power table can be represented by a fixed-point number with integer in 22-bit and fractional in 9-bit. The maximum value of such number will be 4194303.998047. Accordingly, the fixed-point version of power table can be used for multiplication (19) and division (20) with fixed-point X as follows:

$$\text{(fixed)}\,(\text{power\_table}_i * X_{fixed}) >> 9 \quad (19)$$

where $i = 0, 1, \ldots, 127$

$$\text{(fixed)}\left(\frac{1 << (15*3-9)}{X_{fixed} * \text{power\_table}_i}\right) >> 15 \quad (20)$$

where $i = 0, 1, \ldots, 127$

There are some similar cases with root function or square function [22], and those functions can be modified for optimization.

4.4 Precision Optimization

The precision of computations in the study can be improved by modifying the fixed-point format, and some precision optimizations are made during the following steps.

1. Selecting fixed-point format

Since the fixed-point format defined above in Figure 8 is possible to be lower than 15-bit for the fractional bits, the greatest value could be predicated and the value is not larger than 2^{16} in WMA decoding. Hence, the floating-point data can be represented by 15-bit fractional part.

2. Constructing the power table

When building the power table, it can be found that 2 to power of 10 will be always positive in this case as defined in (18). In other words, it is meaningless to keep the sign bit. Thus, we construct the power table by using a fixed-point format with 10-bit fractional part instead of the 9-bit format. As shown in Table 4.1, the additional bit can be used to represent the fractional part of the fixed-point number to improve the fractional accuracy.

3. Exchanging the multiplication order

In the WMA decoding function, it is frequently found that there are multiplications with three floating-point variables. After converting to fixed-point multiplications, the product of two variables multiplications in original order may exceed the represented max value of the fixed-point format defined in Figure 8. In order to solve such an overflow problem, the multiplication order should be changed by firstly selecting two smaller variables in the three variables multiplications, and then multiplying the largest variable with the product.

4. Handling the small value

Although floating-point variables can be converted into fixed-point formats, not all variables can be converted with high accuracy. For instance, by the window function (5), the value of sine function is known in the range $-1 \le \sin \le 1$, therefore we can modify the format in Figure 8 by increasing the fractional part to 25-bit, decreasing the integer-part to 6-bit and keeping the sign bit. Thus, the range of the modified format will be -64.000000 to 63.999999. Similarly, the small value in noise table (6) and the largest value in noise table can be found in (21).

$$\begin{aligned}\text{noise\_table}_{max} &= \text{seed}_{max} * \text{norm}_{max} \\ &= 2147483647 * 3.2261 \cdots e^{-11} = 0.06928 \cdots,\end{aligned} \quad (21)$$

where $\text{norm}_{max} = 1/2147483648 * 1.73205 \cdots * 0.04 = 3.2261 \cdots e^{-11}$, and
$\text{seed}_{max} = \text{integer}_{max} = 2147483647$

The same format can be applied to represent the noise table to improve the precision accordingly. Other small

Table 4.1. Improvement of fractional accuracy

	Unsigned integer	Signed integer
Fractional-part bits	10	9
Precision	2^{-10}	2^{-9}

values in the decoding function can also be represented by such format. We have defined the converting function (22), (23) and multiplication truncation (24) in 25-bit fractional-part format as follows:

$$\text{(fixed) FFP\_CONV}(X_{float}) = \text{(signed integer)}(X_{float} * (1 << 25) + 0.5) \quad (22)$$

$$\text{(fixed) FFP\_CONVI}(X_{integer}) = X_{integer} << 25 \quad (23)$$

$$\text{FFP\_MUL\_TRUNC}(X, Y) = \text{product}(X, Y)_{64\text{bits}} >> 25 \quad (24)$$

where $\text{product}(X, Y)_{64\text{bits}} = X_{32\text{bits}} * Y_{32\text{bits}}$

5. Reducing the table value

As proposed in section 4.3 that we have constructed the power table and square root table for optimizing computational complexity. Since the difference of power of 10 values is too large, the power table size should be reduced from 196 to 60 items. When the power value is greater than $10^{60*0.05}$, the power value is decomposed to $10^{60*0.05} * 10^{x}$ where x can look up in power table. For instance, the calculation $10^{100*0:05}$ can be decomposed as shown in equation (25):

$$10^{100*0.05} = 10^{60*0.05} * 10^{40*0.05} \quad (25)$$

The way of modifying the square root table is identical to power table.

5. Experiment Results and Comparisons

In order to verify the usability of the proposed optimizations for WMA compatible decoder on a non-FPU RISC core, Samsung SBC-2410x is chosen as target embedded testbed in this paper, which equipped with a 32-bit RISC processor [23]. Originally, a UNIX based WMA decoder (e.g. Mplayer [24], FFmpeg [25]) was utilized to play WMA audio on the SBC-2410x embedded system. Unfortunately, the experiment almost leaded to the system crash with the CPU utility rate going over than 100%. Since SBC-2410x with ARM9 core employs limited architecture to reach lower cycles per instruction (CPI), the implementation performance of WMA decoding software, such as Mplayer, is very low. Figure 11 shows the CPU performance after employing the proposed optimization technique in comparison with the original WMA decoder and Modified IMDCT. In fact, when the WMA audio file is played without optimization, the CPU usage rate is over 500% and it nearly made the embedded system crash. After modifying the functions in IMDCT with fixed-point conversions, the CPU utility rate is reduced to about 150%. Even though the experiment result is remarkable, the goal is still unattained. Subsequently, it can be observed that, by applying the fixed-point optimizations to modify the window functions, the

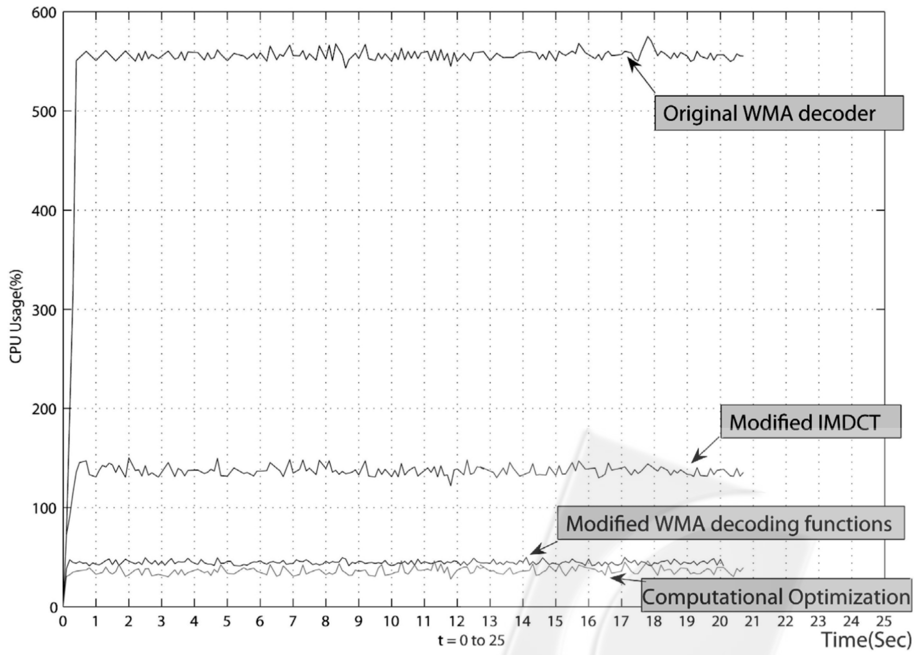


Figure 11. CPU Utility Rate Comparison.

CPU utility rate is reduced to about 45%. Finally, the power and square root function in *math.c* is replaced by fixed-point version to refine our effort.

When evaluating the fidelity of a sequence, the differences value e_i between the original audio values decoded by floating-point version and fast integer version are taken into consideration, as shown in Figure 12. In general, it is difficult to examine the difference on a term-by-term basis. Therefore, the average measures are applied to summarize the information in difference sequence. Firstly, the absolute difference (abs_diff) between the value y_i decoded by fixed-point and x_i by floating-point with all samples N is calculated by formula (26), then the absolute difference and mean squared error (MSE) are calculated by formula (27) and (28) respectively [1].

$$e_i = x_i - y_i, \text{ where } i = 1, 2, \ldots, N \qquad (26)$$

$$\text{Mean abs\_diff} = \frac{\sum_{i=1}^{N} |e_i|}{N} \qquad (27)$$

$$MSE = \frac{\sum_{i=1}^{N} (e_i)^2}{N} \qquad (28)$$

In the simulation of audio quality, four types of WMA music are chosen as the simulation cases. Figure 13(a) to

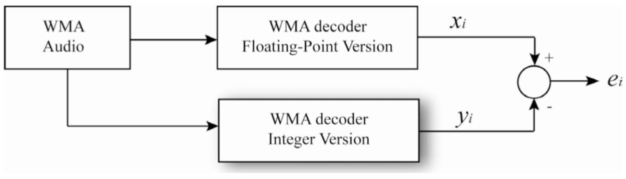
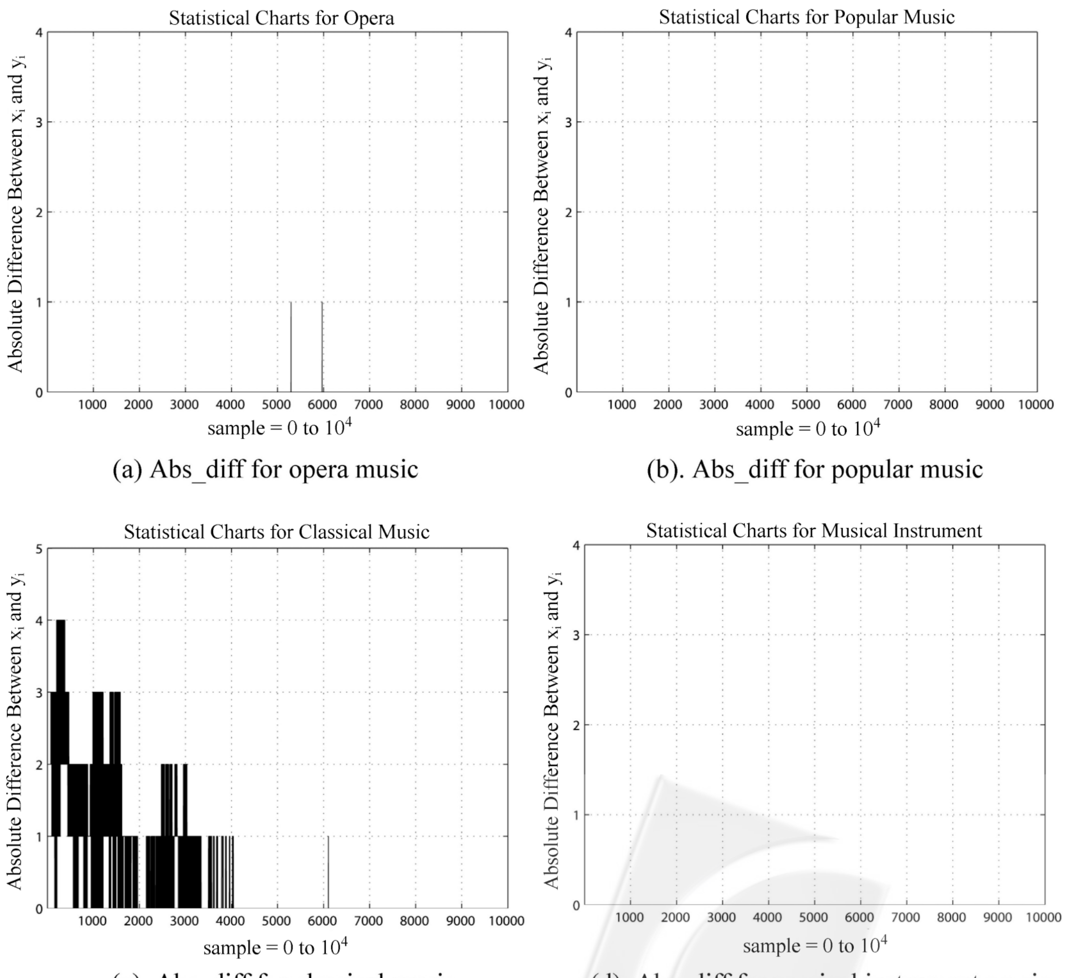

WMA Audio
WMA decoder Floating-Point Version
WMA decoder Integer Version
x_i
y_i
+
-
e_i

Figure 12. Data for Decoding



(a) Abs_diff for opera music

(b). Abs_diff for popular music

(c). Abs_diff for classical music

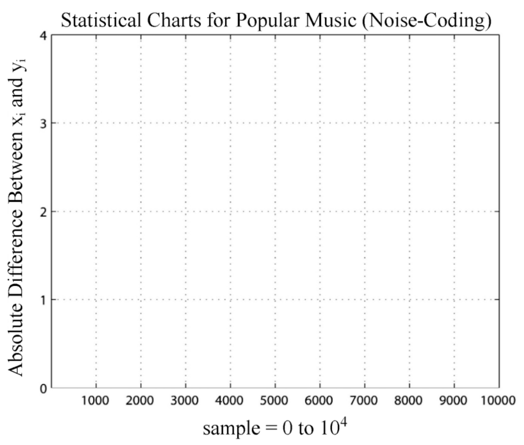
(d). Abs_diff for musical instrument music

Figure 13. The error analysis for decoding different types of music

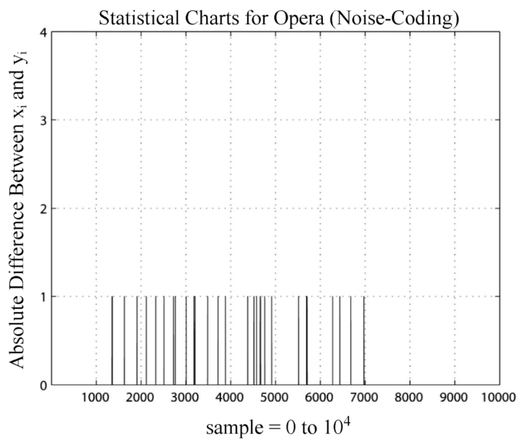
(d) show the absolute difference of x_i and y_i with respect to ten thousand samples in each case. Furthermore, if the bit-rate and sample-rate of WMA music are lower than the thresholds, the noise-coding function can be used to improve the reality of music [25]. Figure 14(a) to (d) shows another four cases of music sampled with the noise-coding function. The results of the fixed-point error simulations associated with the parameters of maximum abs_diff, mean abs_diff and maximum error bits are summarized in Table 4.2. The maximum error bits can be defined by the following formula (29).

$$\text{Max error bits} = \text{the least integer } k\text{, such that } 2^k > \text{Max abs\_diff} \tag{29}$$

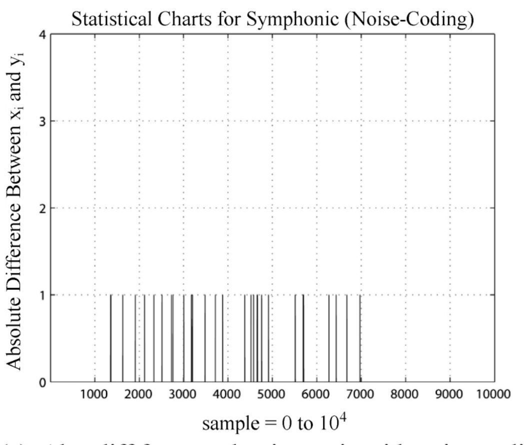
According to the results in Table 4.2, our proposed



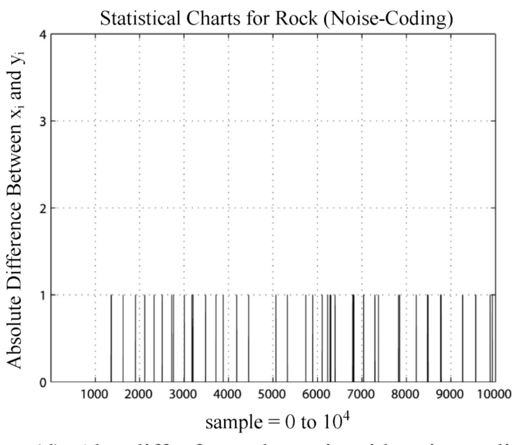
(a). Abs_diff for popular music with noise-coding



(b). Abs_diff for opera music with noise-coding



(c). Abs_diff for symphonic music with noise-coding



(d). Abs_diff a for rock music with noise-coding

Figure 14. The error analysis for decoding different types of music with noise-coding

Table 4.2. Results of error analyses for fixed-point optimization

Music type	Noise coding	Samples	Max abs_diff	Mean abs_diff	Max error bits	MSE
13(a). Opera	No	13942784	1	$2.322 * 10^{-3}$	1-bit	$2.322 * 10^{-3}$
13(b). Popular	No	21508096	1	$2.075 * 10^{-3}$	1-bit	$2.075 * 10^{-3}$
13(c). Classical	No	14417920	4	$1.728 * 10^{-3}$	3-bit	$2.031 * 10^{-3}$
13(d). Musical	No	17715200	1	$2.365 * 10^{-3}$	1-bit	$2.365 * 10^{-3}$
14(a). Popular	Yes	11806720	1	$2.178 * 10^{-3}$	1-bit	$2.178 * 10^{-3}$
14(b). Opera	Yes	14774272	1	$2.322 * 10^{-3}$	1-bit	$2.322 * 10^{-3}$
14(c). Symphonic	Yes	16011264	1	$3.890 * 10^{-3}$	1-bit	$3.890 * 10^{-3}$
14(d). Rock	Yes	28418048	1	$7.216 * 10^{-3}$	1-bit	$7.216 * 10^{-3}$

optimization for WMA compatible decoders results in an average error of only approximately 1-bit.

6. Conclusions

In this paper, we have presented embedded software code optimization methods and optimized the WMA compatible decoder. By the proposed fixed-point function transformations, the purpose that enabling embedded system to play WMA audio files fluently was achieved without an FPU processor. After the optimization process, the WMA compatible decoder only required of about 45% CPU utility rate to decode different types of WMA audio on SBC-2410x with an average of one error-bit. Furthermore, our software achievement made it possible to play WMA as MP3 on several non-DSP embedded systems. It is important to note that the proposed fixed-point optimizations can not only be applied to minimize the computational overhead for audio decoding, but also are suitable for the video decoding with low-cost. In other words, the proposed fast integer optimal approach is helpful to the implementation of an open-source [26] multimedia decoder on a single RISC core embedded system in the future.

References

[1] Soderquist, P., Leeser, M. and Rojas, J.-C., "Enabling MPEG-2 Video Playback in Embedded Systems Through Improved Data Cache Efficiency," *IEEE Trans. Multimedia*, Vol. 8, pp. 81–89 (2006).

[2] Yongseok Yi and In-Cheol Park, "A Fixed-Point MPEG Audio Processor Operating at Low Frequency," *IEEE Trans. Consumer Electronics,* Vol. 47, pp. 779–786, (2001).

[3] Yao, Y., Yao, Q., Liu, P. and Xiao, Z., "Embedded Software Optimization for MP3 Decoder Implemented on RISC Core," *IEEE Trans. Consumer Electronic*s, Vol. 50, pp. 1244–1249 (2004).

[4] You, S. and Hou, Y., "Implementation of IMDCT for MPEG2/4 AAC on 16-bit Fixed-Point Digital Signal Processors," *Proc. 2004 IEEE Asia-Pacific Conf. on Circuits and Systems,* Vol. 2, pp. 813–816 (2004).

[5] Wang, H., Xu, W., Dong, X., Li, C. and Yu, W., "Implementation of MPEG-2 AAC on 16-bit Fixed-Point DSP," *Proc. IEEE Asia-Pacific Conf. on Circuits and Systems 2006,* pp. 1903–1906 (2006).

[6] Lee, K.-H., Lee, K.-S., Hwang, T.-H., Park, Y.-C., Youn, D. H., "An Architecture and Implementation of MPEG Audio Layer III Decoder Using Dual-core DSP," *IEEE Trans. Consumer Electronics*, Vol. 47, pp. 928–933 (2001).

[7] MP3: MPEG-1 Audio Layer3 Home Page, http://www.thomson.com

[8] Xiph Ogg Home Page, http://www.xiph.org/ogg

[9] MAD: MPEG Audio Decoder Home Page, http://www.underbit.com/products/mad

[10] Microsoft Window Media. Home Page, http://www.microsoft.com

[11] Underbit Technologies Home Page, http://www.underbit.com

[12] Geiger, R., Herre, J., Koller, J. and Brandenburg, K., "INTMDCT - A Link Between Perceptual And Lossless Audio Coding," in *Proc. IEEE Int. Conf. Acoustice, Speech, and Signal Processing,* Vol. 2, pp. 1813–1816 (2002).

[13] Geiger, R. and Schuller, G. "Integer Low Dealy and MDCT Filter Banks," in *IEEE Int. Conf. Signal, Systems and Computers,* Vol. 1, pp. 811–815 (2002).

[14] Mu-Huo, C. and Yu-Hsin, H. "Fast IMDCT and MDCT Algorithms – A Matrix Approach," *IEEE Trans. Acoustics*, *Signal Processing,* Vol. 51, pp. 221–229 (2003).

[15] Brenner, N. and Rader, C. "A New Principle for Fast Fourier Transformation," *IEEE Acoustics, Speech & Signal Processing,* Vol. 24, pp. 264–266 (1976).

[16] Mu-Huo, C. and Yu-Hsin, H., "Fast IMDCT and MDCT Algorithms – A Matrix Approach," *IEEE Trans. Signal Processing,* Vol. 51, pp. 221–229 (2003).

[17] Qraintara, S., Chen, Y. and Nguyen, T., "Integer Fast Fourier Transform," *IEEE Trans. Signal Processing,* Vol. 50, pp. 607–618 (2002).

[18] Yusong, Y., Guangda, S., Chunmei, W. and Qingyun, S., "Invertible Integer FFT Applied on Lossless Image Compression," in *IEEE Int. Conf. Robotics, Intelligent Systems and Signal Processing*, Vol. 2, pp. 1219–1223 (2003).

[19] Princen, J. and Bradley, A., "Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation," *IEEE Trans. Acoustic, Speech, and Signal Processing,* Vol. 34, pp. 1153–1161 (1986).

[20] Ki-ll, K., Jiyang, K. and Wonyong, S., "A Floating-Point to Integer C Conveter with Shift Reduction for Fixed-Point Digital Signal Processors," in *Proc. IEEE*

Int. Conf. Acoustice, Speech, and Signal Processing, Vol. 4, pp. 2163–2166 (1999).

[21] Kim, S., Kum, K. and Sung, W., "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," *IEEE Trans. Circuits and Systems-II: Analog and Digital signal Processing,* Vol. 45, pp. 1455–1464 (1998).

[22] Qraintara, S. and Krishnan, T., "The Integer MDCT and Its Application in the MPEG Layer III Audio," in *Proc. IEEE Int. Symp. Circuits and Systems,* Vol. 4, pp. 301–304 (2003).

[23] Samsung Home Page, http://www.samsung.com/tw

[24] C Standard Library Home Page, http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html

[25] FFmpeg Multimedia System Home Page, http://ffmpeg.mplayerhq.hu, (2008).

[26] The Open Source Initiative Home Page, http://www.opensource.org/docs/osd

Manuscript Received: Dec. 16, 2007
Accepted: Jun. 27, 2008