A full-compliance MP3 decoder using DSP

Lawrence K. W. Law, Ph.D.
Marketing Manager
Wireless Infrastructure Systems Division
Networking & Computing Systems Group, Asia
Motorola Semiconductors Hong Kong Ltd

Kenneth K. C. Lee
Department of Computer Science
City University of Hong Kong

Abstract
This paper describes the techniques used in the non-linear dequantization process of MP3 (MPEG audio, Layer III) decoding, with Motorola Star*Core DSP (MSC8101). The method introduced here is more than five times faster than calling traditional mathematics library, yet achieves a high level of accuracy that conforms to the ISO/IEC 13818-4 standard [2]. Memory consumption of this method is considerably low and is very suitable for DSP implementation of the MP3 decoder in real-time.

1.0 Introduction

MPEG-1 (Moving Pictures Experts Group) is a standard for compressing digital video and audio, at a combined bit-rate of 1.5Mbit/sec. The standard is divided into several parts and the third part (11172-3) specifies the standard for audio compression [1]. The audio compression standard consists of three layers with different complexity and performance, named as Layer I, II and III. The Layer III standard (usually referred to as "MP3"), is the most complex among the three layers. Like Layer I & II, layer III makes use of "Sub-band Synthesis" for transforming audio signal. On top of it, it introduces Huffman coding to reduce the bit-rate of audio frames and also use non-linear quantization to improve the sound quality. DSP implementation of MP3 player has been wildly adopted in applications such as exchanging music in internet, hand-held music players, PDAs, Hi-Fi and transportation audio systems nowadays. In order to implement the MP3 in real-time running simultanously with other applications, a powerful DSP has to be used.

The StarCore 140 in Slide 4 is a low cost, low power, high performance, high flexibility programmable general purpose fixed-point DSP core with the 3rd generation DSP Archnitecture that efficiently deploys a novel Variable Length Execution Set (VLES) execution model utilizing maximum parallelism by allowing multiple Data and Address ALUs to execute multiple operations in a single clock cycle. A Data Arithmetic Logic Unit (DALU) performs arithmetic and logical operations on data operands in the Star*Core 140 core. The Star*Core 140 has 4 Arithmetic & Logic units in the DALU. Four instances of a single-cycle Multiplier-Accumulator (MAC) Unit with automatic saturation capability and four instances of a Bit Field Unit (BFU), each with a 40-bit barrel shifter capable of executing a variety of single-bit and multi-bit logic and shift operations.

2.0 MP3 Decoding Algorithm

Slide 2 shows the outline of MP3 decoding process. After seeking the "Sync Word" (which identifies the start of frame) and reading the audio frame header, The decoder should first fetches in the "Side Information", which contain information about audio block type, Huffman tables, gain and scale factors. After reading the scale factors of different scale-factor-bands, the decoder decodes the frequency samples using the Huffman decoding scheme. Then, the frequency samples are dequantized. Unlike Layer I & II, the dequanzition process uses non-linear scale. After that, the samples of both channels undergoes stereo processing (both Middle-Side-Stereo & Intensity-Stereo), Antialiasing. And after the IMDCT process, Sub-band Synthesis process is applied to the sub-band samples to yield the PCM audio signal.

The most computational intensive parts are the "Sub-band Synthesis", "IMDCT", "Dequantization" and the "Huffman Decoding". Fast algorithms for "Sub-band Synthesis" and "IMDCT" has been discussed in many papers [3]. In this paper, fast method for sample dequantization will be presented.

3.0 Sample Dequantization

The dequantization process of MP3 decoding can be written as:

$$xr_i = sign(is_i) * |is_i|^{4/3} * gain * scale\_factor$$

where $\mathbf{is}_i$ is the input sample and $\mathbf{xr}_i$ is the dequantized sample. The problem is "How to calculate $\mathbf{x}^{4/3}$? (where **x** is an integer)". Although mathematics libraries are available to different kinds of DSP, there are simply too time-consuming. Even using optimized assembly code, the ***power()*** mathematics routine takes about 115 cycles per calculation. For a stereo music stream at 48kHz, only this part consumes more than 11 MIPS. Hence this method is not suitable for real-time DSP implementation.

An alternative is to use pure table lookup. As the range of $\mathbf{is}_i$ is bounded to range 0..8207, by storing 8207 entries (of $\mathbf{x}^{4/3}$)

into memory, one can get the output sample easily. However, the memory consumption is huge. Assume each entry is stored in 32 bits. The whole table takes about 33kbytes of memory. Even 24-bit word is used, the table is still very large (25 kbytes).

Here, we use a new method to solve the problem. What's difficult about the formula is that the "power" term (4/3) is not integer. If the power is integer, we can calculate it easily by successive multiplication. The main idea of our method is to use polynomial of x to estimate the curve of $y=x^{4/3}$

$$y' = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + \dots\dots\dots$$

So, the program only need to store the coefficients $a_0, a_1.....a_n$, instead of storing the whole 8207 entries of the curve. The order of polynomial (number of terms) depends on the required accuracy. The greater the number of terms, the more accurate is the estimation.

To use a single curve for estimation is difficult, so, the curve is broken down into regions and each part is estimated individually (shown in Slide 11). Dividing the curve into regions is advantageous. First of all, one can keep the estimation error low by setting some threshold value: Observe the estimation error along the x-axis, (from 0 towards 8207), if the estimation error gradually increases. We can stop the current estimation curve and use a new estimation curve for the remaining part of the curve (Slide 12). This method allows trade-off between table size and the estimation error.

Also, dividing the curve into regions enables uneven range size. Observed that when x is small, the curve is more "curved" and the curve is more "linear" when x is large. So, when x is small, one can divide the curve more precisely (e.g. use range size of 32); and when x is large, one can use larger range size (e.g. use size of 512).

Another advantage of dividing the curve is that one can use polynomials of different orders to estimate different parts of the curve. For example, when x is large, the error percentage is relatively small, so, one can use a polynomial of order 2 ($a_0+a_1x+a_2x^2$) to estimate that part. When **x** is small and the curve is more "curved", one can use polynomial of 3rd or 4th order. This makes our method very flexible, one can add in new polynomial terms or new regions if more precision is needed. And one can delete terms or regions if the memory constraint is tight (with the cost of some precision).

4.0 Range Identification

So, the method for getting $\mathbf{x}^{4/3}$ is straight forward. After fetch in input **x**, the program find whether **x** is in region 0, 1, 2, or ...etc. After identifying the region of the input **x**, use the polynomial of that range to estimate $\mathbf{x}^{4/3}$. The question is how to break down the curve and how to identify the region of x. A simple approach is to set arbitrary upper bound and lower bound and use a series of if-then-else to check the range. For example, we break the curve to range: (0-1000), (1001-2000), (2001-3000), ...etc. Then the pseudo code would be like:

```
If (0<x<1000)
    {use polynomial1}
ElseIf (1001<x<2000)
{use polynomial2}
ElseIf (2001<x<3000)
{use polynomial3}
Else if ...... etc.
```

The major problem is that the code contains a large number of *if-then-else*. In assembly language, this is equivalent to a lot of *compare* and *jump*. Each *jump* instruction takes about 3 to 4 cycles, which is more expensive than normal arithmetic operations. (which usually uses 1 cycle only). Even worse, the greater the number of regions, the greater the number of *if-then-else* and the worse the performance.

Here we propose a better method: which is, to use the range boundary that is aligned to power of 2 (2^n, where n=5..10). (Slide 15 shows the range used in our prototype of MP3 decoder)

Note that in each region, all number has the same number of leading zeroes. (when presented as binary number). So, in the implementation on Motorola Star*Core DSP, one can use the CLB (Count Leading Bits) instruction to identify the range of input **x**. Many other DSP, such as the DSP563xx family should have similar instruction for counting bits. The advantage of this method is fast and direct. Only 2 cycles, we can find the range where the input x is located. Another advantage is that this method uses "Unven Division". When **x** is small (0-32), the range size is small (32); when x is large (4096-8191), the range size is large (4096). This allows a good banlance between the table size and the percentage error. Figure 7 shows the table used in one of our implementation. Note that when **x** is small, the range size is smaller and the "order" of polynomial is higher. The reason is mentioned before. Using the table in Figure 7, the mean percentage error is about 0.0295%.

5.0 Accuracy and Performance

ISO/IEC 13818-4 standard states the accuracy requirement for limited-accuracy and fully-accurate MPEG Audio decoders. To be an full-accurate decoder, the RMS (Root Mean Square) between the reference and the decoded signal should be less than 2-15/sqrt(12) relative to full-scale, when decoding a given "sine-sweep". Also, the maximum absolute different should be at most 2-14 relative to full-scale. For a full-scale of 64k (16-bit output), the Mean RMS should be less than 0.577, and the maximum absolute difference should be no more than 4. Using our method, the RMS is 0.880 and the maximum difference is 4.949. Which is much better than limited-accuracy decoders and come very close to the fully-accurate standard. (Figure 8). Also, the method is very efficient. It takes a maximum of 22 cycles to calculate $\mathbf{x}^{4/3}$, which is 5 times faster than calling mathematics library (115 cycles). There is only 31 (3*9+4) entries in the lookup table (of coefficients), which is much smaller than traditional table lookup (8207 entries).

6.0 Higher Accuracy

In the method described above, the index for table-lookup is obtained by counting leading zeroes of **x** (yields 10 regions). Actually, the index can be derived by other methods, say, by counting leading zeros of functions or polynomial consisting x. For example, in actual implementation, our decoder counts the leading zeros of $\mathbf{x}^2$. (yields 20 regions).

The reason for using $\mathbf{x}^2$ is that: In a 32-bit word, **x**, the number of leading zero = **x** have range 0..8207, hence $\mathbf{log_2(x)}$ has range 0..13. There are <u>at most 14 ranges</u>

$\mathbf{x}^2$ have range 0..67354849, $\mathbf{log_2(x^2)}$ has range 0..26. There are <u>at most 27 ranges</u>

Hence, by squaring **x**, <u>the number of region is nearly doubled</u>. This results in a more accurate curve (and larger table size). Figure 9 compares the two partitioning methods. Note that when using $\mathbf{x}^2$, the curve is more precisely divided and hence more accurate. Figure 10 shows the actual table used.

Using the new method, the mean percentage error is reduced to 0.0047%. The new function still uses 22 cycles only (because when evaluating polynomial, $\mathbf{x}^2$ is calculated anyway). The new table uses 61 entries of coefficients, which is still smaller than pure table lookup (8207 entries).

Using this implementation, the Mean RMS is 0.356 and the Maximum difference is 1.680, which is better than the fully-accurate conformance standard proposed by ISO. As a reference, we also implement another version, using a full lookup table of 8207 entries. The achieved RMS is 0.326 and the Maximum difference is 1.086. It shows that the estimation method is highly accurate and close to optimum.

7.0 Hybrid Scheme

As mentioned above, this method is a flexible, many other partition methods are possible. Look at the initial partition method (by counting leading 0 of **x**), when **x** is large, the range size is large. So, for a few applications, it may find the accuracy of those regions not high enough.

On the contrary, for the partitioning method that counts leading 0 of $\mathbf{x}^2$. When **x** is small, the range size is very small. The high accuracy of those regions may be unnecessary for some applications. So, one can combine both schemes to yield a hybrid scheme (Slide 22).

When **x** is small (e.g. $\mathbf{x}<1024$), the range is partitioned according to leading 0 bits of x;

When **x** is large ($\mathbf{x}>1024$), the range is partitioned according to leading 0 bits of $\mathbf{x}^2$.

With this arrangement, the range size will not be too small (unnecessarily accurate) for small **x** and the range will not be too large (not accurate enough) for large **x**.. This gives a good trade-off between table size and accuracy.

References

1. ISO/IEC 11172-3, Information technology - Coding of moving pictures and associated audio for digital storage media at up to 1.5Mbit/s. Part 3 - Audio
2. ISO/IEC 13818-4, Information technology - Generic coding of moving pictures and associated audio. Part 4 - Conformance.
3. Tadashi Sakamoto, Maiko Taruli, and Tomohiro Hase, "A Fast MPEG-Audio Layer III algorithm for 1 32-bit MCU", IEEE transactions on consumer electronics, Vol. 45, No.3, Aug 1999

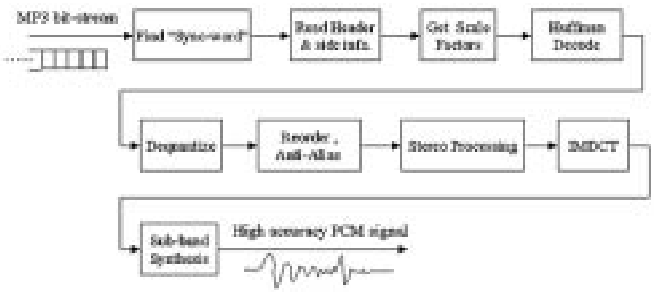
Authors' contact details

Lawrence K. W. Law, Ph.D.
Wireless Infrastructure Systems Division
Networking & Computing Systems Group, Asia
Motorola Semiconductors Hong Kong Ltd.
Silicon Harbour Center
2 Dai King Street
Taipo Industrial Estate
Tai Po, New Territories, Hong Kong
Phone: (852) 2666 8961
Fax: (852) 2663 3277
E-mail: R21718@email.mot.com

Kenneth K. C. Lee
Department of Computer Science
City University of Hong Kong

MP3 decoding

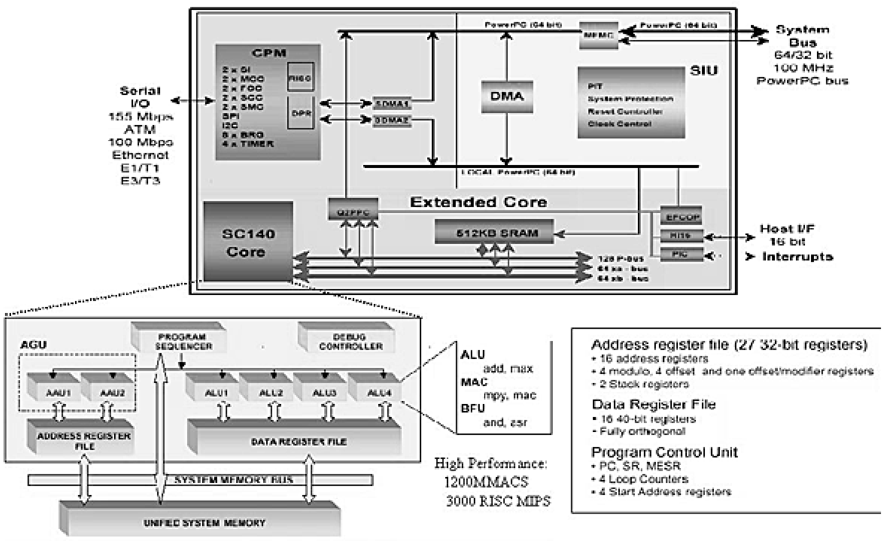
- MPEG (Motion Picture Experts Group) is a standard for compressing digital audio and video, where MP3 is the Layer III of the MPEG Audio standard.
- The decoding of MP3 audio stream is a complex, multi-stage process as shown below.



MP3 Player Market Place

- Standalone portable MP3 player market
- Mobile phone with MP3 player
- PDA with MP3 player
- Digital Audio Hi-Fi System
- Audio System in transportation (car)
- VCD/DVD player with MP3 function.

Architecture of MSC8101 Star*Core DSP



MSC8101 Star*Core DSP Highlight

- Uses top of the line PowerPC process technology
  - > HiPerMOS 6 - advanced copper interconnect
  - > 0.13-0.1 micron L-effective physical gate length
  - > High speed: 300 MHz core
  - > Very low power dissipation: 0.5W @ 1.5V for the whole device
- Leverage on proven Motorola IP
  - > PowerPC bus interface
  - > MPC8260 Memory Controller
  - > MPC8260 Communications Processor Module (CPM)
  - > FSRAM memory cell - 512KB of on chip memory
- DSP inside, PowerPC outside
  - > Commonality in Hardware design
  - > Glueless interfaces with world leading PowerPC architecture
  - > Reuse of drivers
  - > Can be used as PowerPC companion processor

MSC8101 Star*Core DSP Highlight (Cont.)

- Programmable protocol machine (uses a 32 bit RISC engine)
  - > Supports a wide variety of protocols
  - > Reuse of proven MPC8260 microcode
  - > Upgradable for future protocols
  - > Potential customer specific protocols
- Network connectivity to standard backbones
  - > 155 Mbps ATM SAR (Utopia bus) supporting AAL 0/1/2/5
  - > 10/100 Mbps Ethernet
  - > Up to four E1/T1 interfaces or one E3/T3 and one E1/T1
  - > HDLC support up to T3 rates or 256 channels

Instruction: Flexibility for MP3 Decoder

- **MAC: 4 different MAC instructions in one cycle**
  - **mpy d0,d1,d2 macr -d0,d3,d4 impy d5,d6,d7 mac d2,d4,d5 move.l** `(r0)+,d8 move.l (r1)+, d9`
- ALU: 4 different ALU instructions in one cycle
  - **add d0,d1,d2 cmpgt d3,d4 maxm d7,d8 sub d4,d5,d6**
- BFU: 4 bit manipulation instructions in one cycle
  - **and d0,d1 asrr #7,d3 asll d2,d7 ror d4**
- Many flexible combinations possible...
  - **mpy d0,d1,d2 cmpgt d3,d4 asll d2,d5 insert #5,#4,d6,d7**

Dequantization of samples

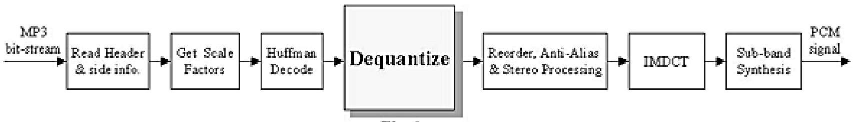


Fig. 1

- After Huffman Decoding, samples need to be dequantized.
- Samples are divided into bands, each band have its own gain & scale factor.
- Dequantization uses non-linear equation:

Short Blocks:

$$xr_i = sign(is_i)*|is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain-210-8*subblock\_gain)} * 2^{-(scalefac\_multiplier*short\_scale\_factor)}$$

Long Blocks:

$$xr_i = sign(is_i)*|is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain-210)} * 2^{-(scalefac\_multiplier*(long\_scale\_factor+pretab\_value))}$$

Fig. 2

The Challenge:

$$xr_i = sign(is_i)*|is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain-210-8*subblock\_gain)} * 2^{-(scalefac\_multiplier*short\_scale\_factor)}$$

- The core of dequantizing is to calculate $|Integer(0..8207)|^{\frac{4}{3}}$ and $2^{-\frac{1}{4}*Integer}$
- The dequantization process should be efficient and minimize table/ROM usage
- The dequantization must be accurate enough to satisfy ISO/IEO 13818-4 conformance standard. (For full conformance, Mean RMS should be <0.57 and Max. absolute different should be <4 for 16-bit output)
- The accuracy of $|Integer(0..8207)|^{\frac{4}{3}}$ and $2^{-\frac{1}{4}*Integer}$ affects the overall performance.
- We focus our effort to develop a good method to estimate the value of |*is*|^4/3 here.

Dequantization of samples - Existing methods

How to Calculate/Estimate $|Integer(0..8207)|^{\frac{4}{3}}$?

1. Calculate during dequantization, with C/Assembly build-in Mathematics library.
(Sample decoder : Decoder by ISO MPEG Audio Subgroup Software Simulation Group)

=> **Unreasonable MIPS consumption** (acceptable for PC, but not for DSP)
(e.g. The "power" calculation routine by Lucent uses about 115 cycles per calculation)

=> **Considerable code size.** The "power" calculation routine is usually large
(e.g. The "power" calculation routine by Lucent is about 1.2k byte large)

2. Stores the values in memory table, either as "constants" or calculated once during decoder initialization phase.
(Sample decoder : "MA play 1.2", "LAME 3.51", "MPG-123 ver 0.59r")

=> **Unreasonable Memory Consumption** (acceptable for PC, but not for DSP)
(Table size = 8208, each entry = 32 bits, Total = 32832 bytes)

Dequantization of samples

How to Calculate/Estimate $\left|Integer(0..8207)\right|^{\frac{4}{3}}$?

The curve of $y = x^{4/3}$ can be estimated by series:

$$y' = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + \ldots\ldots\ldots$$

Where y' is the estimate value of y and $a_0, a_1, a_2, \ldots$, are some real constants

We only need to stores $a_0, a_1, a_2, a_3, a_4, \ldots\ldots\ldots$, instead of storing whole 8208 entries

To use a single series to estimate the whole curve is difficult, hence the curve is broken down into smaller regions (according to value of x) and each region is estimated individually.

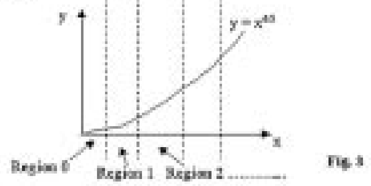
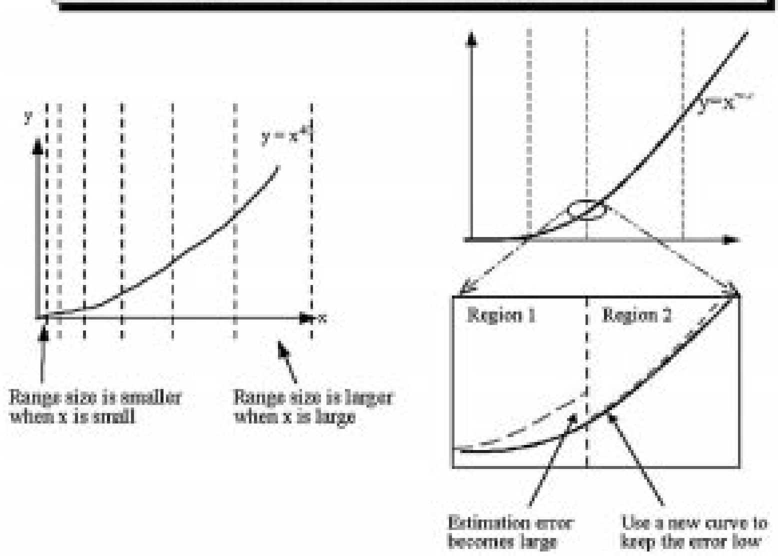


Fig. 8

Advantage of non-uniform region breakdown



Advantages to break down the curve in regions:

- **Keep the estimation error low**. (e.g. When the error exceed threshold, we can use a new curve starting from that point, so that the error is low again)
- **Uneven range size is possible**. (e.g. when x is large, error percentage is relatively small, hence the range size can be larger)
- **Regions can be estimated with different "orders"** (i.e. number of terms).
  (e.g. when x is small, we can use "4th-order": $a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$
  when x is large, we can use "2nd-order" only : $a_0 + a_1x + a_2x^2$")
- **Can trade-off between the number of coefficients and the error percentage.**
  (the greater the number of coefficients, the higher the accuracy)
- The curve is broken down into regions, each region having different set of coefficients.
- During estimation of $x^{4/3}$, we should:

  1. Find the range where x is in. 2. Use the coefficient set of that range for calculation.

The efficiency of step 1 (Finding the range of x) largely depends on how we partition the curve.

Method to Break Down the Curve

1. **Set arbitrary upper-bound and lower-bound** of range and use a sequence of "if-then-else" to select the correct set of coefficients.

   => Not used because of relatively large code-size and MIPS consumption

E.g. Range = (0..1000),(1001..2000),(2001-4000),(4001-8207)

```
If (x<=1000) Then
        use Coefficient-Set 0
Else if (x<=2000) Then
        use Coefficient-Set 1
Else if (x<=4000) Then
        use Coefficient-Set 2
Else
        use Coefficient-Set 3
End-If
```

If-then-else is actually a set of **compare and jump.** Such ***"Jump-if-true"*** and ***"Jump-if-false"*** instructions requires 3-4 cycles, so, for the worse case, 10-20 cycles are wasted just to struggle for the correct range

In this example, worse case requires 3 comparisons

The situation is worse if the number of ranges is large!

2. Use boundaries which are aligned to powers of 2.

Range	Range Size	Number of leading 0	26 - # of leading 0
0-31	32	32-27	---
32-63	32	26	0
64-127	64	25	1
128-255	128	24	2
256-511	256	23	3
512-1023	512	22	4
1024-2047	1024	21	5
2048-4095	2048	20	6
4096-8191	4096	19	7
8192-8207	16	18	8

Fig. 4

Notice that the numbers between 2 boundaries has the same number of leading zero.

Hence we can use the "CLB" (Count leading bits) instruction of the DSP to determine which region is the number in.

After adjusting the return value of **CLB**, the number can be used directly for table-lookup

Very fast in determining the range !!

Another advantage is that it performs **uneven partitioning:**
- The range size is small when x is small (e.g. 32-63),
- The range size is large when x is large (e.g. 4096-8191)

- One can estimate each region with any method he like. And one can use any number of terms per estimation. (The higher the "order", the more accurate is the estimation)

In our invention, each region use "2nd-order", and range 0..31 uses "3rd-order".
(Because when x is small, percentage error is relatively large, so we need better estimation)

	width	Range	a0	a1	a2	a3
part0	32	0-31	-0.43670023	1.32857277	0.09763591	-0.00129873
part1	32	32-63	-17.42554112	3.15853316	0.01745636	
part2	64	64-127	-44.46108777	3.99174848	0.01093090	
part3	128	128-255	-111.99462997	5.02876297	0.00688745	
part4	256	256-511	-282.60871996	6.33802143	0.00433591	
part5	512	512-1023	-712.61087563	7.98871268	0.00273058	
part6	1024	1024-2047	-1805.67994957	10.07657685	0.00171548	
part7	2048	2048-4095	-4527.92298185	12.66028146	0.00108327	
part8	4096	4096-8191	-11426.54030096	15.98200384	0.00068193	
part9	16	8192-8207	-11427.36842840	15.96871367	0.00068173	

Fig. 5

The mean error percentage = 0.0295%

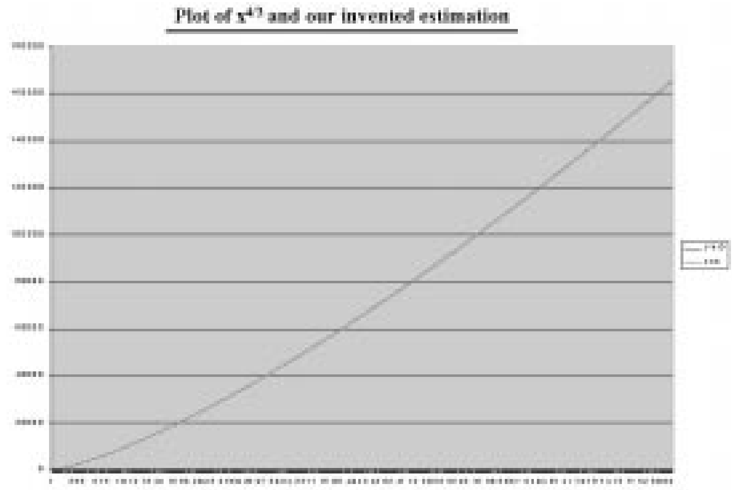


Fig. 6

• The above graph shows that our estimation is highly accurate !!

ISO/IEC 13818-4 standard states the accuracy requirement for MPEG Audio decoders

To be an full-accurate decoder, the RMS (Root Mean Square) between the reference and the decoded signal should be less than **2^{-15}/sqrt(12)** relative to full-scale, when decoding a given "sine-sweep". Also, the maximum absolute different should be at most **2^{-14}** relative to full-scale

For a full-scale of 64k (16-bit output), the Mean RMS should be less than **0.577**, and the maximum absolute difference should be no more than 4.

Using the invented **estimation** method, the RMS is **0.880** and the max. different is **4.949** which is quite close to "full-accurate" standard

It is, however, qualified for "limited-accuracy" decoder (0.577 <RMS < 9.237)

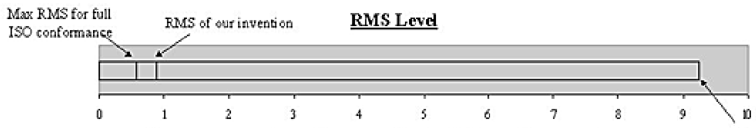


Fig. 7

When decoding songs and music, no audible different is noticed.

Total table size: (1x4 + 9x3) x 32 bits = **128 bytes**
Maximum cycles per calculation: **22 cycles**

Method to achieve higher accuracy

In the previous method, index of table is calculated by counting **leading zeros of x**. (**10** regions obtained).

Actually, the index can be derived by other methods, say, by counting leading zeros of functions/polynomial consisting **x**. For example, in actual implementation, our decoder counts the **leading zeros of x^2**. (**20** regions obtained)

Why x^2 ?

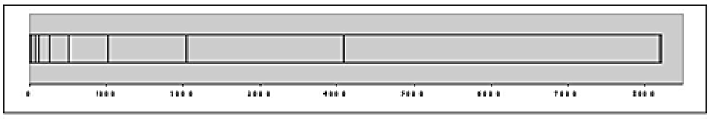
In a 32-bit word, x, the number of leading zero = $31 - \lfloor \log_2(x) \rfloor$

x have range 0..8207, hence log2(**x**) has range 0..13. There are at most 14 ranges
x^2 have range 0..67354849, log2(**x^2**) has range 0..26. There are at most 27 ranges

Hence, by squaring x, the number of region is nearly doubled. This result in a more accurate curve (and a larger table size)

The mean error percentage = 0.0047%

partitioning
according to x
(Limited accuracy)
Fig. 8



Partitioning
according to x^2
(Full conformance)
Fig. 9

Total table size: (1x4 + 19x3) x 32 bits = **244 bytes** ("3rd-order" for region 0..15, other uses "2nd-order")

Max. cycles per calculation: **22 cycles** (same, because x^2 has to be calculated anyway)

Achieved Mean RMS: **0.356** Max absolute different: **1.680**
Which is <u>qualified for a fully accurate decoder</u>

As a reference (to find the best achievable), we also tried to use a lookup table of size 8208 (32-bit)
Achieved Mean RMS: **0.326** Max absolute different: **1.086**

It shows that our estimation method is highly accurate

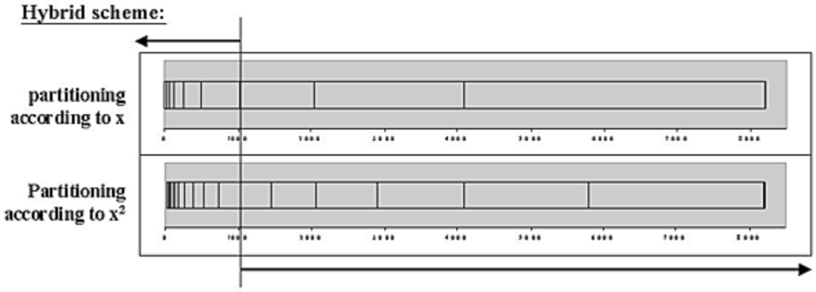
<u>Table used in actual implementation for conformance:</u> (by counting leading zeros of x^2)

	Range	Width	a0	a1	a2	a3	x^2	Number of leading 0 of x^2
part 0	0-15	16	-8.3294	1.2048	0.12749	-0.00287	0-225	32-24
part 1	16-22	7	-5.2730	2.3351	0.03214		256-484	23
part 2	23-31	9	-8.6459	2.6426	0.02609		529-961	22
part 3	32-45	14	-13.6126	2.9597	0.02081		1024-2025	21
part 4	46-63	18	-22.4848	3.3557	0.01556		2116-3969	20
part 5	64-90	27	-35.6771	3.7633	0.01238		4096-8100	19
part 6	91-127	37	-56.3247	4.2213	0.00984		8281-16129	18
part 7	128-181	54	-89.9113	4.7449	0.00779		16384-32761	17
part 8	182-255	74	-139.8959	5.2994	0.00624		33124-65025	16
part 9	256-362	107	-227.8070	5.9811	0.00490		65536-131044	15
part 10	363-511	149	-359.6069	6.7096	0.00389		131769-261121	14
part 11	512-723	212	-589.1873	7.5264	0.00309		262144-522729	13
part 12	724-1023	300	-898.7189	8.4322	0.00247		524176-1046529	12
part 13	1024-1448	425	-1374.4841	9.3929	0.00199		1048576-2096704	11
part 14	1449-2047	599	-2211.9563	10.5676	0.00157		2099601-4190209	10
part 15	2048-2896	849	-3608.0846	11.8591	0.00125		4194304-8386816	9
part 16	2897-4095	1199	-5418.9991	13.2255	0.00100		8392609-16769025	8
part 17	4096-5792	1697	-8874.3137	14.9561	0.00078		16777216-33547264	7
part 18	5793-8191	2399	-14195.4578	16.8184	0.00062		33558849-67092481	6
part 19	8192-8207	16	-19268.6213	18.1847	0.00053		67108864-67354849	5

Fig. 10

Hybrid Scheme for Partitioning

<u>Hybrid scheme:</u>



- When **x** is small (e.g. **x**<1024), the range is partitioned according to leading 0 bits of **x**;
- When **x** is large (**x**>1024), the range is partitioned according to leading 0 bits of $\mathbf{x^2}$.
- gives a good trade-off between table size and accuracy.