Using MMX™ Instructions to Implement Audio Echo Sound Effects

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Note: Please be aware that the software programming article below is posted as a public service and may no longer be supported by Intel.

Copyright © Intel Corporation 2004
* Other names and brands may be claimed as the property of others.

Using MMX™ Instructions to Implement Audio Echo Sound Effects

March 1996

CONTENTS

1.0 INTRODUCTION
- 1.1 Waveform Audio File Format and Data Structure

2.0 ECHO SOUND EFFECTS
- 2.1 Echo Sound Effect Algorithm
- 2.2 C Implementation of Algorithm
- 2.3 MMX™ Implementation of Algorithm
  - 2.3.1 Shift Factor
  - 2.3.2 Unoptimized MMX™ Code
  - 2.3.3 Optimization Procedure
  - 2.3.4 Optimized MMX™ Code
  - 2.3.5 Unrolled MMX™ Code

3.0 PERFORMANCE GAINS

4.0 CONCLUSION

1.0 INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multiple-data (SIMD) instructions which are designed to allow many pieces of information to be processed with a single instruction, providing parallelism that greatly increases performance.
Audio data is generally stored in 8-bit or 16-bit integer data types. Depending on the application, the new SIMD instructions could perform up to 8 calculations in parallel on 8-bit audio data.
This application note presents examples of code that exploit these SIMD instructions to add echo to existing audio data.

1.1 Waveform Audio File Format and Data Structure

The Audio Echo Sound Effect application described in this document operates on 8-bit audio data extracted from files stored in the Microsoft/IBM Waveform Audio File Format (also known as RIFF WAVE format). Files of this format commonly have a ".wav" file extension.

The sound data stored in the WAVE file consists of samples represented in pulse code modulation (PCM) format. For 8-bit audio samples each sample is described by an unsigned 8-bit integer from 0 (0x00) to 255 (0xFF), with the midpoint, or origin at 128 (0x80). Audio data with the value of 0x80 represents no sound. In a single-channel WAVE file, samples are stored consecutively. For stereo WAVE files, samples are interleaved:

Table 1: Format of Stereo Samples

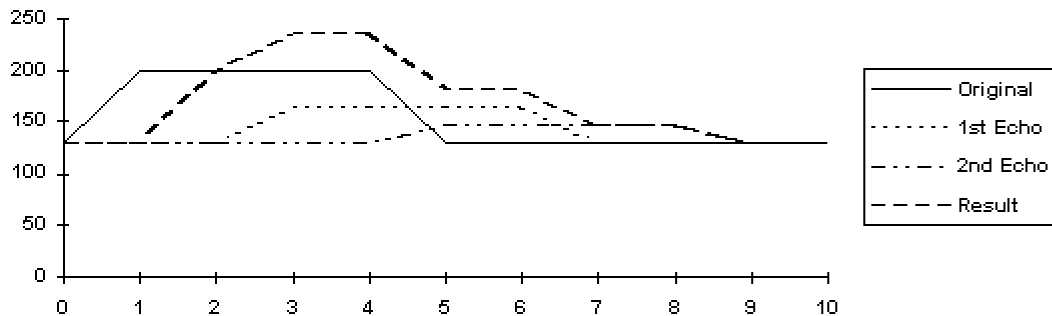
Sample 1		Sample 2		Sample 3	
Left Channel	Right Channel	Left Channel	Right Channel	Left Channel	Right Channel
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5

An array of bytes or unsigned chars is used to store audio samples extracted from a WAVE file. To facilitate aligned memory accesses, the array is padded with elements containing 0x80 (no sound) to make the size of the array a multiple of 8 bytes.

2.0 ECHO SOUND EFFECT

An echo is the repetition of a sound created by sound waves reflecting from a surface. We can simulate an echo in a WAVE file by adding an attenuated, time delayed version of the waveform audio data to the original data. Figure 1 shows an example of an echo with a delay of two sample time periods.

Figure 1: Superposition of Echoes



2.1 Echo Sound Effect Algorithm

A simple mathematical representation of the echo is the following equation: |

Figure 2: Echo Equation

$$Y[n] = X[n] + \sum_{i=1}^{NumEchos} Gi * X[n - i * Delay]$$

where ,

X[n] = nth sample from the original audio waveform

NumEchos = The number of echoes

G_i = The attenuation factor for the ith echo

Delay = The number of samples time periods to delay the echo

Y[n] = nth sample from the resultant audio waveform

In the example shown in the chart above, NumEchos = 2, Delay = 2, and Gi = 1/(2*i).

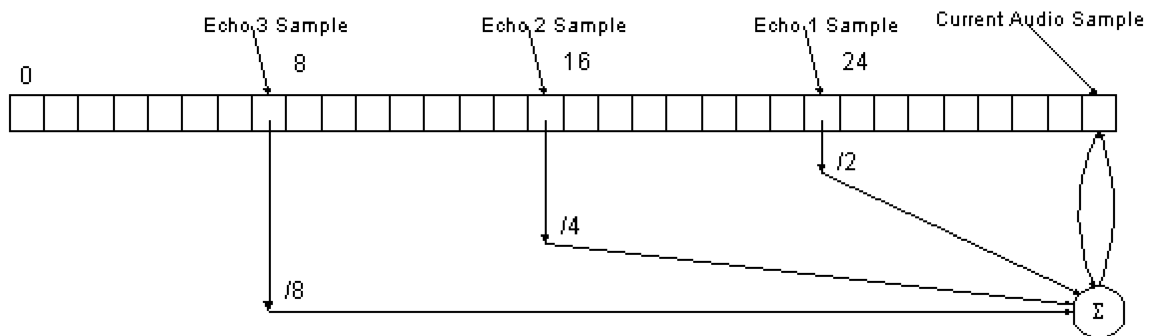
For Equation 1 to be valid, the waveform audio samples will need to be converted from an unsigned integer (range 0 to 255, midpoint 128) to a signed integer (range -128 to 127, midpoint 0) by inverting the most significant bit of the sample.

2.2 C Implementation of Algorithm

To implement Equation 1, we will need two arrays if we process the data starting with the first sample (n=0) and continuing to the end of the array. However, by processing the data in the reverse order, we can accomplish the same results with only one array.

For example, assume that we have a 32-byte array (buffer) containing audio samples, the delay is 8, and the number of echoes is 3. Processing of the array would start at last element of the array (buffer[size-1]) and continue down to buffer[Delay]. The first echo would be attenuated by 1/2, the second echo by 1/4, etc. Refer to Figure 1.

Figure 3: Calculating Echo Effect



The echo function can be implemented using the following sequence of C code:

Code Listing 1: Simple C Implementation of Echo Sound Effects

```
void Echo8(unsigned int size, unsigned int Delay, unsigned int NumEcho,char* buffer) {
      unsigned int LastEl;
      LastEl = (size/8)*8;                          // align data access on eight byte
                                                    // boundaries
      unsigned int k;
      // outer loop
      for (unsigned int i=LastEl-1; i>Delay; i--)  // compare occurs
      {                                             // (LastEl-Delay) times
            buffer[i]   = buffer[i]^0x80;           // convert data to unsigned
            k=1;
            // inner loop
            // compare occurs at least (NumEcho)*(LastEl-Delay) times
            while ((k<=NumEcho)&&(Delay*k <=I))    // process all echoes, but stay
            {                                       // in boundaries of array
                // accumulating the effects of the echoes
                  buffer[i] = buffer[i] + ((signed char)(buffer[i-Delay*k]^0x80) >> k);
                  k++;
            }
            buffer[i]   = buffer[i]^0x80;           // convert back to unsigned
      }
}
```

Referring to Code Listing 1, we can see that the operations performed in the inner loop are performed a total of at least $(LastEl - Delay) * NumEcho$ times. The code can be modified facilitate its conversion to

MMX™ by partially unrolling its loops. Unrolling the code also results in faster code because it reduces the average cost-per-loop of Jump instructions.

The following code segment partially unrolls the loop by performing the required calculations on 8 bytes sequentially on each iteration. This code is for illustrative purposes only and does not represent the best possible method.

Code Listing 2: Unrolled C Routine

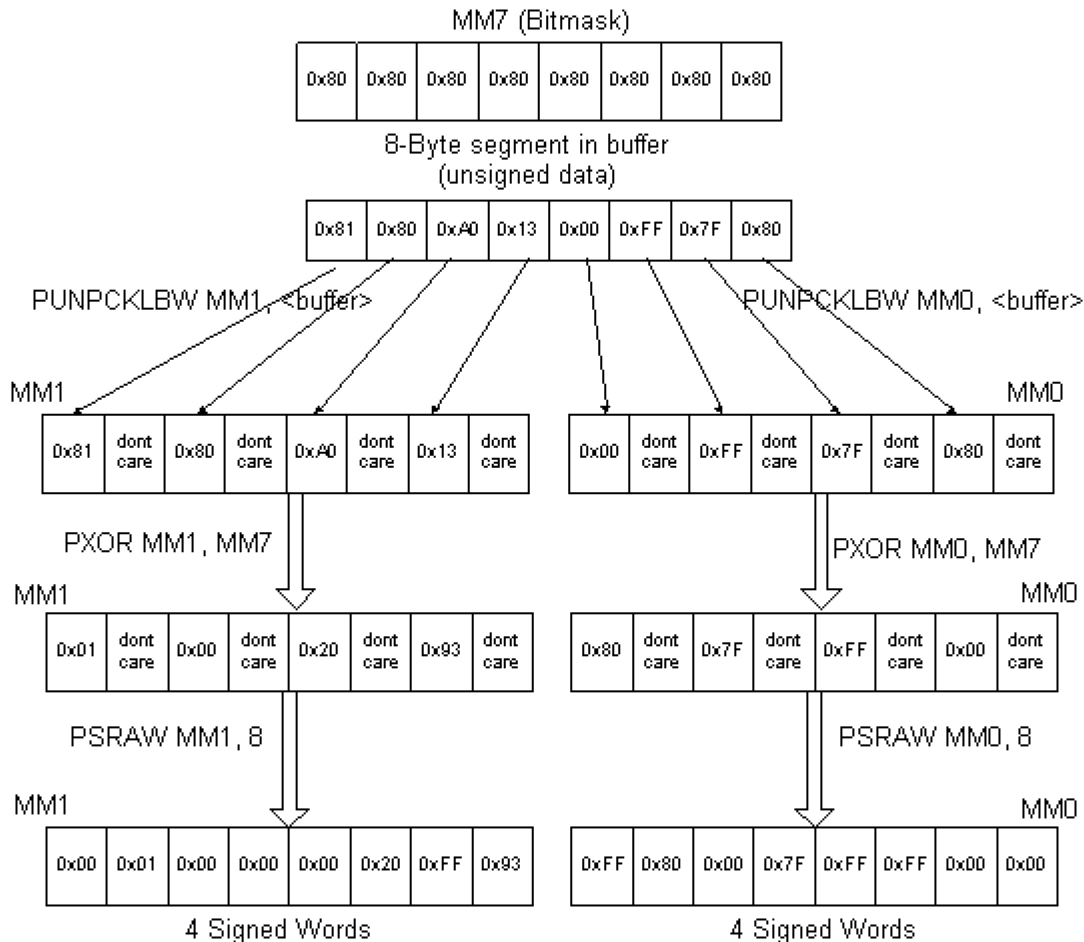
```
// outer loop
// loop unrolled
// compare occurs (LastEl-Delay)/8 times
for (unsigned int i=LastEl-8; i>Delay; i-=8)
{
      buffer[i]   = buffer[i]^0x80;
      buffer[i+1] = buffer[i+1]^0x80;
      buffer[i+2] = buffer[i+2]^0x80;
      buffer[i+3] = buffer[i+3]^0x80;
      buffer[i+4] = buffer[i+4]^0x80;
      buffer[i+5] = buffer[i+5]^0x80;
      buffer[i+6] = buffer[i+6]^0x80;
      buffer[i+7] = buffer[i+7]^0x80;
      k=1;
      // inner loop
      // compare occurs at least NumEcho*(LastEl-Delay)/8 times
      while ((k<=NumEcho)&&(Delay*k <=i))
      {
            buffer[i]   = buffer[i]   + ((signed char)(buffer[i  -Delay*k]^0x80) >> k);
            buffer[i+1] = buffer[i+1] + ((signed char)(buffer[i+1-Delay*k]^0x80) >> k);
            buffer[i+2] = buffer[i+2] + ((signed char)(buffer[i+2-Delay*k]^0x80) >> k);
            buffer[i+3] = buffer[i+3] + ((signed char)(buffer[i+3-Delay*k]^0x80) >> k);
            buffer[i+4] = buffer[i+4] + ((signed char)(buffer[i+4-Delay*k]^0x80) >> k);
            buffer[i+5] = buffer[i+5] + ((signed char)(buffer[i+5-Delay*k]^0x80) >> k);
            buffer[i+6] = buffer[i+6] + ((signed char)(buffer[i+6-Delay*k]^0x80) >> k);
            buffer[i+7] = buffer[i+7] + ((signed char)(buffer[i+7-Delay*k]^0x80) >> k);
            k++;
      }
      buffer[i]   = buffer[i]^0x80;
      buffer[i+1] = buffer[i+1]^0x80;
      buffer[i+2] = buffer[i+2]^0x80;
      buffer[i+3] = buffer[i+3]^0x80;
      buffer[i+4] = buffer[i+4]^0x80;
      buffer[i+5] = buffer[i+5]^0x80;
      buffer[i+6] = buffer[i+6]^0x80;
      buffer[i+7] = buffer[i+7]^0x80;
}
```

2.3 MMX™ Implementation of Algorithm

The MMX implementation uses the same general algorithm. However, each iteration through the loop operates on sets of 8-bytes (as in the unrolled C routine, but with 4-bytes in parallel).

The following example shows how the initial data is read from the buffer and converted from unsigned bytes into signed words. The data is converted to words because the packed shift instructions operate on words, double words and quad words only.

Figure 4: Reading and Converting to Signed Data



2.3.1 Shift Factor

The shift factor is the value needed to shift the upper byte of a word to the lower byte of the word. In the example shown in Figure 3, the shift factor is 8 (PSRAW MM0, *8*).

In attenuating each echo, we divide the echo element by powers of 2. The first echo is divided by 2, the second by 4, the third by 8, etc. This attenuation is most efficiently done by using the packed shift right instruction. Both of these shifts can be accomplished in one instruction by shifting 8 + (number of echo) bits. For example, the first echo is shifted a total of 8+1 bits, second echo is shifted 8+2 bits.

2.3.2 Unoptimized MMX™ Code

In all of the following MMX implementations of the echo function the local variable BITMASK is defined as a quadword with the value of 0x8080808080808080.

The following assembly code sequence implements the echo function in its original, unoptimized form. Line spacing is used to indicate logical blocks and do not indicate super-scalar pairing.

Code Listing 3: Unoptimized MMX Echo Function

```
START_MMXECHO:
        mov             esi, LastEl
        mov             edi, mybuffer           ; pointer to the buffer
        movq            mm7, dword ptr BITMASK

        sub             esi,00000008            ; pointer to current sample set
        jmp             START_L_1
LOOP_1:
        sub             esi,00000008            ; i = i - 8
START_L_1:
        mov             ebx, 8
        movd            mm6, ebx        ; intialize shift factor
        punpcklbw       mm0, [edi + esi] ; get low byte
        pxor            mm0, mm7        ; convert to signed
        psraw           mm0, mm6        ; realign (convert to word)
        punpckhbw       mm1, [edi + esi] ; get high byte
        pxor            mm1, mm7        ; convert to signed
        psraw           mm1, mm6        ; realign (convert to word)
        mov             ecx, 0                  ; k = 0
        mov             eax, Delay              ;
        mov             edx, esi        ; copy index to buffer
        sub             edx, eax        ; get to delay
        jc              END_LOOP_2
        jmp             START_L_2
LOOP_2:                                         ; loops on number of echoes
        inc             ecx                     ; k = k + 1
        add             eax, Delay              ; goto next delay
        mov             edx, esi        ; copy index to buffer
        sub             edx, eax        ; get to delay - check for array bounds
        jc              END_LOOP_2
        inc             ebx                     ; shift factor = 8 + k
        movd            mm6, ebx

START_L_2:
        punpcklbw       mm2, [edi + edx] ; get low byte of echo k
        pxor            mm2, mm7        ; convert to signed
        psraw           mm2, mm6        ; apply shift factor
        paddsw          mm0, mm2        ; accumulate "low byte" words sums in to mm1
        punpckhbw       mm3, [edi + edx] ; get high byte of echo k
        pxor            mm3, mm7        ; convert to signed
        psraw           mm3, mm6        ; apply shift factor
        paddsw          mm1, mm3        ; accumulate "high byte" words sums in to mm1
        cmp             ecx, NumEcho            ; loop back if k (ecx) <= NumEcho
        jb              LOOP_2
END_LOOP_2:
        packsswb mm0, mm1          ; pack low and high words into bytes
        pxor            mm0, mm7        ; convert back to unsigned
        movq            [edi + esi], mm0 ; store result back to buffer
        cmp             esi, Delay
        jb              END_L_1
        jmp             LOOP_1
END_L_1:
```

2.3.3 Optimization Procedure

The speed of echo sound effect routines, like other assembly code routines, can benefit greatly from optimization. This echo sound effect routine benefited most from these optimizations:

- Pairing

Re-ordering instructions to make better use of both of the Pentium™ processor super-scalar pipelines. MMX™ instructions are similar to scalar integer instructions, in that they can be executed two at a time, if certain microarchitecture constraints are met. These constraints are codified into "pairing rules."

- Data alignment

Ensuring that memory reads and writes do not cross their natural alignment boundaries. For instance, the 64-bit MOVQ memory accesses should be done to 8-byte-aligned memory addresses. If a boundary is crossed, the routine stalls, while the processor makes an extra memory access.

- Loop unrolling

A loop is unrolled by writing it out N times, with each new, longer loop executed only 1/Nth as many times. Loop unrolling helps pair instructions at the (original) loop edges, and reduces the average cost-per-loop of Jump instructions. The inner loop of this routine is a good candidate for loop unrolling.

The general procedure was to first create a straight-forward echo effects routine, which was easy to understand and debug, then to optimize that routine in successive steps. All functional verification testing was done using a C++ program which had the ability to read, write and play WAVE files. Performance testing and optimization was done using a separate C program which simply called the various routines using a sample 800 byte data set. Correct operation was verified throughout the optimization process by playing the resulting audio data and comparing the results to a similar routine coded in C.

Data alignment was easily achieved by declaring the critical arrays first, in Microsoft Visual C. A more robust method would be to explicitly check data alignment at run-time, and re-align them then, if necessary.

The optimization effort was concentrated primarily in the loops. Any cycle time saved in the loops is greatly magnified, as the loops execute on the order of thousands of times. Code outside of this loop (subroutine call/return, parameter retrieval, constant matrix setup) is executed only once per call, so did not merit much optimization effort.

Each stage of optimization consisted of:

- Re-ordering instructions for pairing and stall avoidance
- Verifying functional correctness
- Using simulation tools to verify that the desired pairings and timings were accomplished

2.3.4 Optimized MMX™ Code

The following assembly code sequence is the optimized version of the MMX echo function. The lines are spaced to indicate super-scalar pairing.

Code Listing 4: Optimized MMX Echo Function

```
START_MMXECHO:
        mov             esi,LastEl              ; pointer to end of buffer
        mov             edi, mybuffer           ; pointer to the buffer
        movq            mm7, dword ptr BITMASK

        sub             esi,00000008            ; pointer to current sample set
        jmp             START_L_1
LOOP_1:
        sub             esi,00000008            ; i = i - 8
START_L_1:
        mov             ebx, 8
        punpcklbw       mm0, [edi + esi] ; get low byte (lb)
        punpckhbw       mm1, [edi + esi] ; get high byte (hb)
        pxor            mm0, mm7        ; convert lb to signed
        mov             ecx, 0                  ; k = 0
        mov             eax, Delay              ; first delay
        movd            mm6, ebx        ; initialize shift factor
        pxor            mm1, mm7        ; convert hb to signed
        psraw           mm0, mm6        ; realign lb (convert to word (lbw))
        mov             edx, esi        ; copy index to buffer
        psraw           mm1, mm6        ; realign hb (convert to word (hbw))
        sub             edx, eax        ; get to delay
        jc              END_LOOP_2
        jmp             START_L_2
LOOP_2:                                         ; loops on number of echoes
        add             eax, Delay              ; goto next delay
        mov             edx, esi        ; copy index to buffer
        inc             ebx                     ; shift factor = 8 + k
        sub             edx, eax        ; check bounds
        jc              END_LOOP_2
        inc             ecx                     ; k = k + 1
        movd            mm6, ebx        ; mm6 <= shift factor

START_L_2:
        punpcklbw       mm2, [edi + edx] ; get low byte of echo k
        punpckhbw       mm3, [edi + edx] ; get hi byte of echo k
        pxor            mm2, mm7        ; convert echo k lb to signed
        pxor            mm3, mm7        ; convert echo k hb to signed
        psraw           mm2, mm6        ; apply shift factor
        psraw           mm3, mm6        ; apply shift factor
        paddsw          mm0, mm2        ;accumulate "low byte" words sums in mm0
        paddsw          mm1, mm3        ; accumulate "hi byte" words sums in mm1
        cmp             ecx, NumEcho            ; loop back if k (ecx) <= NumEcho
        jb              LOOP_2
END_LOOP_2:
        packsswb mm0, mm1       ; pack low and hi words into bytes
        pxor            mm0, mm7        ; convert back to unsigned
        movq            [edi + esi], mm0 ; store results back to buffer
        cmp             esi, Delay
        jb              END_L_1
        jmp             LOOP_1
END_L_1:
```

2.3.5 Unrolled MMX™ Code

The performance of this function can be further enhanced by unrolling the inner loop (LOOP2). For example, rewrite the code so when NumEcho is 1, one section of code is used, when NumEcho is 2, another section of code is used, and so on. As the code is written now, the inner loop performs two compares and branches for each echo (one for array boundary checks, one for loop termination) per iteration of the outer loop (LOOP1). When the loop is unrolled, only the array boundary check is performed thus reducing the expensive compare and branch instructions by a half. The code segment on the following page is an example of the section of code used to perform two echo calculations:

Code Listing 5: Unrolled MMX Technology Echo Function

```
TWO_ECHOS:
; get two echo delays
; 1st echo in mm2, mm3, modify, accumulate in mm0, mm1
; 2nd echo in mm4, mm5, modify, accumulate in mm0, mm1
; For pairing purposes, some registers are pre-initialized
;***********************************************************************
; 1st echo
       mov            eax, Delay          ; get to delay (Delay = 1 Delay)
       sub            edx, eax            ; get to delay in buffer

       jc             END_TWO_ECHO
       mov            ebx, 9                       ; shift factor for first echo = 8+1

       movd           mm6, ebx            ; mm6 <- shift factor

       punpcklbw      mm2, [edi + edx]    ; get low byte of echo 1
       punpckhbw      mm3, [edi + edx]    ; get hi byte of echo 1
       pxor           mm2, mm7            ; convert echo 1 lb to signed
       pxor           mm3, mm7            ; convert echo 1 hb to signed
       psraw          mm2, mm6            ; apply shift factor to lb
       psraw          mm3, mm6            ; apply shift facoor to hb
       paddsw         mm0, mm2            ; accumulate "low byte" word sums in mm0
       paddsw         mm1, mm3            ; accumulate "hi byte" word sums in mm1
;***********************************************************************
; 2nd echo
       add            eax, Delay          ; goto next delay (Delay = 2 Delay)
       mov            edx, esi            ; copy index to buffer
       sub            edx, eax            ; get to delay
       jc             END_TWO_ECHO
       inc            ebx                          ; shift factor = 8 + 2
       movd           mm6, ebx            ; mm6 <- shift factor
       punpcklbw      mm4, [edi + edx]    ; get low byte of echo 2
       punpckhbw      mm5, [edi + edx]    ; get hi byte of echo 1
       pxor           mm4, mm7            ; convert echo 2 lb to signed
       pxor           mm5, mm7            ; convert echo 2 hb to signed
       psraw          mm4, mm6            ; apply shift factor to lb
       psraw          mm5, mm6            ; apply shift facoor to hb
       paddsw         mm0, mm4            ; accumulate "low byte" word sums in mm0
       paddsw         mm1, mm5            ; accumulate "hi byte" word sums in mm1
;***********************************************************************
;finished with echoes
;
```

3.0 PERFORMANCE GAINS

All performance analysis was done using Intel Vtune 2.0 Beta 3.0. Each routine operated on the same data set:

- 800 element byte array
- Number of echoes = 4
- Delay = 48 sample time periods

Table 2 show the results of the dynamic and static analysis of each routine.

Table 2: Vtune Performance Analysis Results

	Vtune Analysis				
	Non-MMX™ C Routine[1]		**MMX Routine**[2]		
	Normal	Unrolled	Unoptimized	Optimized	Optimized & Unrolled
Dynamic Analysis					
# Instructions Run	45698	16554	8442	8442	6105
Total Cycle Count	*41186*	*28322*	*7660*	*6712*	*4809*
CPI	0.90	1.71	0.91	0.80	0.79
%Pairing	95%	58%	37%	71%	71%
Static Analysis					
# Instructions	46	86	56	56	91
Total Cycles	76	145	51	46	70
%Pairing	78%	53%	46%	64%	68%
Performance					
Performance Gain Compared to Unrolled C[3]	NA	1 X	3.4 X	4.2 X	5.9 X
Performance Gain Compared to Unoptimized MMX[4]	NA	NA	1 X	1.1 X	1.6 X

Notes:

1 C routines were compiled using Microsoft Visual C++ with the compiler options set to produce blended code and optimization set to maximum speed

2 MMX Routine assembled with MASM 6.11d

3 Performance Gain = (Total Cycle Count of Normal C Routine) / Total Cycle Count

4 Performance Gain = (Total Cycle Count of Unoptimized MMX Routine) / Total Cycle Count

Table 2 illustrates the significant performance gain of even unoptimized MMX code over the similar C routine.

4.0 CONCLUSION

This application note has demonstrated how MMX technology can be successfully used to accelerate an echo effects algorithm. The MMX instruction set was shown to map well to audio transformation calculations. Optimized code was almost twice as fast as naive (unoptimized) code.

While the exact requirements of particular audio applications will vary, MMX technology is generally applicable. Intel's MMX technology provides an opportunity for a new level of software audio performance on PCs.