



Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

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 Optimized Motion Compensation for MPEG1 Video Playback

March 1996

CONTENTS

- 1.0. INTRODUCTION
- 2.0. MPEG DESCRIPTION
 - 2.1. Video Frame Structure
 - 2.2. Full/Half Pel
- 3.0. CODE DESCRIPTION
 - 3.1. MC Kernels
 - 3.2. Code Macros
- 4.0. CODE FUNCTIONALITY
 - 4.1. Misalignment Avoidance
 - 4.2. Cache Considerations
 - 4.3. Cache Locality
 - 4.4. Color Component
 - 4.5. Relation with iDCT Code
 - 4.6. Shift Tables
- 5.0. HALF-PEL SHORTCUTS
 - 5.1. Avoiding Accuracy Losses
 - 5.2. Speed
- 6.0. REFERENCES

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents examples of code that exploit these instructions. Specifically, these instructions illustrate how to use the new MMX technology to perform motion compensation (MC) for Moving Pictures Expert Group (MPEG) Video playback. The performance gain relative to traditional Intel Architecture (IA) code can be attributed to the MMX instructions used to avoid misalignment problems.

2.0. MPEG DESCRIPTION

2.1. Video Frame Structure

The data stream used by the example code supplied at the end of this application note is provided by the MPEG encoder. The encoder provides the data in three types of frames: B, P and I frames. Each frame is composed of macro blocks (MB) and can be built from a different combination of blocks. These rules apply to the block combinations:

- I frames are composed only of I-type MB
- P frames are composed of I-type and P-type MB
- B frames are made up of I-type, P-type and B-type MB

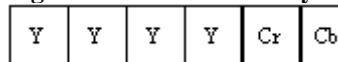
Motion compensation (MC) is performed only in P-type and B-type macro blocks.

Each MB is composed of 6 blocks:

- 4 8x8 pixel Y blocks that compose a 16x16 pixels block
- 1 8x8 pixel Cr block
- 1 8x8 pixel Cb block

This composition is shown in Figure 1.

Figure 1. Macro Block Layout.



Macro blocks (MBs) are based on reference video blocks. The reference blocks can be either a past video block, a future video block, or both. The original MB is compared to the reference MB and two elements are calculated by the encoder motion vectors (MV) to indicate the MB motion horizontally and vertically relatively to the reference block and a delta. The delta is the difference in pixels between the two blocks encoded with Discrete Cosine Transform (DCT). P-type MBs are then built based on one reference frame while B-type MBs are built upon two reference frames.

Motion vectors point to the referenced past and/or future blocks and, in the attached code, are supplied in the MPEG bit stream as a pointer. These pointers may be located anywhere in the referenced frame, so in some cases they might not be aligned on MB memory boundary. The kernels calculate a new MB which is aligned on MB memory boundary (address mod 8 = 0).

2.2 Full/Half Pel

The MPEG encoder selects a MV that minimizes the delta between the reconstructed MB and the actual needed value to a minimum number of bits. This is done with a Discrete Cosine Transform (DCT). In order to decrease the number of bits needed to describe the delta, the MV can be specified either in half pixels (half-pel) units or full pixels (full-pel), both in the horizontal and vertical plane. When the horizontal vector specifies a half-pel, an average is calculated between every pixel and the pixel on its right. In the case of a vertical half-pel, the average is calculated between every pixel and the pixel just below it. If both the horizontal and vertical vectors specify half-pel, the four adjacent pixels are averaged.

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

3.0. CODE DESCRIPTION

The MC code is supplied in the form of two sets of Microsoft Macro Assembler (MASM) macros. One set of macros is the unscheduled code the assembly instructions are written in natural data dependent order. This version can be used to understand the algorithm. The second set of macros contains the optimized scheduled code.

A few shortcuts from the MPEG standard were taken in the example code. The reasons and justification are described in Section 5.1.

3.1. Motion Compensation Kernels

MC is done in two types of MB: P-type and B-type, each with four possible combinations of half-pel (horizontal and vertical axes). Each such combination has a separate kernel for the Y component and the CrCb Component. Out of the 16 MC kernels possible, five Y component kernels are supplied: all four possible half-pel combinations for the P-type MB and a full-pel/full-pel (Full/Full) case for the B-type MB. Each kernel is expected to be used only once in the source code (no effort to localize labels were done).

3.2. Code Macros

The macro names are short by convention (four or five letters): the first letter is the MB type (P or B), the second is the component (Y or CC), the third and fourth are the half-pel indications for the horizontal and vertical vectors, respectively (F for full, H for half). In addition, macros ending with UNS are the UNScheduled macros.

In all cases, all the MMX registers are assumed to be scratch. Also, four integer registers are assumed to be scratch: `eax`, `ebx`, `ecx`, and `edx`.

The other three integer registers are set as follows:

`esi` is a pointer to the upper left pixel in the Y reference MB (or one of the two in case of computing a B-type MB). This address may be unaligned.

`edi` is a pointer to the upper left corner of the calculated Y MB. This address is always aligned.

`ebp` is the second reference pointer when the calculated MB is of B type. This address may be unaligned.

All the macros receive two parameters as input:

`ROW_SIZE` the number of bytes between one pixel and the pixel just below it. `ROW_SIZE` is assumed to be evenly divisible by 8 ($\text{mod } 8 = 0$) in order to avoid misalignment penalties, and to be a complete multiplication of 16 to increase cache locality.

`cont_label` the address to which the execution will return upon completion of the macro.

The data structures are assumed to be defined in the source module: `srcount`. "Shift Right Count", `slcount`. "Shift Left Count".

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

4.0. CODE FUNCTIONALITY

While the main task of the code is to average pixels, other issues are described in this section to bring out some of the techniques implemented in the accompanying code.

4.1. Misalignment Avoidance

The primary innovation when dealing with the main data stream is avoiding misalignment, since misalignment costs an extra three clocks on the Pentium® processor. The data stream is such that the destination MB is always aligned but the reference MB may be anywhere in the frame (i.e. may be misaligned).

The code takes advantage of the MMX instruction operation, `psrlq` (pack shift right logical quad), which takes 1 clock count as opposed to the Pentium processor integer shift operation (which takes four clocks).

This can be illustrated with an easy example. In the case of a horizontal half-pel, one 16-pixel line is averaged with the same line shifted right by one pixel. (Assuming that `esi` is pointing to the first line and is aligned.)

Example 1 illustrates a possible way to shift the data by one pixel. This example assumes that the 16 pixels data is already read into the `MM1` and `MM2` MMX registers.

Start by reading two quad words, starting at address `[esi+1]`, and `[esi+9]`. This uses the Pentium processor's misaligned read capability. The penalty of three cycles per read is eight cycles for two reads done together.

Example 1. Averaging Two Pixels

```
movq mm3, mmword ptr [esi+16]    ;read [esi+16] into mm3
                                  ;remember that esi+16 is also aligned
movq mm4, mm1                    ;duplicate mm1 into mm4.
movq mm5, mm2                    ;duplicate mm2 into mm5
movq mm6, mm2                    ;duplicate mm2 again into mm6.
psrlq mm4, 8                     ;Shift mm4 (the 1st quad word) right by 8.
psllq mm5, 56                    ;Shift mm5 left by 56
                                  ;the least significant pixel is at the higher byte.
por mm4, mm5                     ;OR mm4 and mm5 together
                                  ;the first datum shifted by one pixel.
psrlq mm6, 8                     ;repeat line 5, 6 and 7 for mm6 and mm3
psllq mm3, 56                    ;to get the second datum.
por mm6, mm3
```

All together, 10 MMX instructions are used. In this simple example, the gain is small, but in a more complex case, the savings will be even greater. For example, when a division by two or by four is required (which can be implemented by a shift), the two shifts can be combined. The division by four is required in the case of Full/Full half-pel. (See Section 4.6, Shift Tables.)

4.2. Cache Considerations

Generally, three copies of the frame are kept simultaneously: past, future and the current frame. However, if the current frame is a B-type frame, it may be assumed that the data will not be read again or will be used for reference. Therefore, the code can be arranged so that the data will never destroy the cache. This should be done by calculating small sections of the B-type frame and writing directly into the graphic

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

card's memory. Doing so will increase the chances of other frame types (which will be read again as references) to remain in the L2 cache.

4.3. Cache Locality

In order to achieve cache localization, the Y component of the data should be arranged consecutively and in full multiplications of 16. To furthermore improve localization, the buffer should start on a cache line boundary (32 bytes). This will also solve the Mb alignment requirement.

In Example 4, the stride between one line to the next is assumed to be constant and marked `ROW_SIZE` in the macros. This constant can be easily replaced with a variable held in a register with very small extra performance cost.

4.4. Color Component

This application note does not include code examples for Cr and Cb motion control. However, the methods of building the buffers for them are relevant. Both the Cr and Cb blocks are sub-sampled by 2X in horizontal and vertical axes. It is advisable to interleave the Cb and Cr to keep the number of cache lines to minimum. A Cr pixel followed by a Cb pixel is a good layout, for example. Since both components are referenced by the MV together, the pixels can simply be summed in skips of one pixel on the horizontal axis when needed by the half-pel information.

4.5. Relation with iDCT Code

Apart from the MC, the encoder also passes inverse Discrete Cosine Transform (iDCT) delta data. If iDCT delta data is present, it should be added together with the MC data. Since more than eight bits per pixels are needed, this addition must be done in parallelism of four. Berkeley's MPEG player saves storing this intermediate MC data by immediately adding the iDCT results together. This, however, has the drawback of having to deal with smaller blocks of 8x8 bits (the size of iDCT data) for the MC part. Having to do MC in 8x8 blocks has the side effect of having to read part of the reference pixels twice.

A possible approach is to prepare all iDCT data in MMX registers, while filling zeros for missing iDCT data, and then doing the addition on a 16x16 block. While this may increase the number of cycles required to prepare the data, it avoids the need to read the reference pixels twice.

4.6. Shift Tables

In most cases the reference pointer (`esi`) will not point to an aligned address. The example code uses several lookup tables to calculate the needed shift count, based on the result of $(esi \bmod 8) - 1$. The following table is an example for Shift Right Count (`srcount`):

Example 2. Lookup Table

<code>srcount DB</code>	8	16	24	32	40	48	56
<code>srcount DB</code>	56	48	40	32	24	16	8

For example, if the least significant bits of `esi` are 011, the second entries will be selected and the shift counts will be 16 and 48. The code executed will be similar to the Error! Reference source not found., but the shift count will be replaced according to the tables. For accessing the data, the three least significant bits of `esi` will be masked out, so all accesses are aligned.

The lookup tables described in Figure 2 are one of five table pairs. Each table holds five or six entries. Altogether, there are less than 100 values, each value is less than 64 and can be expressed in eight bits. If

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

a quad word had been assigned for each instead, they would use 800 bytes. Instead, the tables use less than 60 bytes. This saves a lot of space while using only a few extra cycles.

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

5.0. HALF-PEL SHORTCUTS

There are many shortcuts that can be applied with half-pel to increase speed while minimally decreasing accuracy. At the extreme, half-pel information can be totally ignored. Alternatively, when required to average four pixels, only the diagonal two can be averaged. Another alternative is averaging only two pixels out of the four.

The MPEG standard requires that accumulation be done in full precision. When averaging, dividing by two, (or four in the case of Full/Full), if the result has a fraction which is greater than, or equal to half it should be rounded up. The example code reduces accuracy for calculating an averaged pixel by accumulating pixels of 8 bits in 8-bit data. This means that each pixel is divided by two (or four) before the accumulation. So when averaging two pixels the least significant bit of each pixel is lost. When averaging four pixels the two least significant bits of each pixels are lost.

The resulting error varies depending on the case: In the half/full and full/half cases, a value of one in 3 out of four cases is lost. Thus a 0.75 value is lost on average. In the half/half case, up to 3 values can be lost, and 1.47 value is lost on average.

5.1. Avoiding Accuracy Losses

While the following technique is not in use in the example kernels supplied, the loss of accuracy can be avoided by expanding each pixel to 16 bits for calculation and then reducing it back to 8 bits. This means giving up the parallelism of 8 (8 instructions at once) and moving to a parallelism of four.

In the following example (Example 3), it is assumed that `MM1` and `MM2` hold the two 8x8-bit pixels to be averaged and that `mask1` holds `x0101010101010101`.

Example 3. Two Pixel Averaging (8 bit)

```
movq mm3, mm1                ;calculate in mm3 the ORed 8x
por mm3, mm2                 ;LS bits of the two input
pand mm3, mmword ptr mask1
pandn mm1, mask1             ;clear out the LS bits
pandn mm2, mask1             ;divide each pixel by 2, add them together
psrlq mm1, 1
psrlq mm2, 1
paddusb mm1, mm2
paddusb mm1, mm3             ; add the rounding
```

Applying the same technique when averaging four pixels is more complex (Example 4). In such a case, it is assumed that `MM1`, `MM2`, `MM3` and `MM4` hold the four 8x8-bit pixels to be averaged and that `mask3` holds `x0303030303030303`, and `val2` holds `x0202020202020202`.

Example 4. Four Pixel Averaging (8 bits)

```
movq mm5, mmword ptr val2    ; initialize mm5 (rounding bit accumulator) to 2
movq mm6, mm1
pand mm6, mmword ptr mask3   ; mm6 holds 2 LsBits of 1st input
paddusb mm5, mm6             ; accumulate rounding bits
pandn mm1, mmword ptr mask3  ; clear out 2 Lsbits in the input
psrlq mm1, 2                 ; divide the input by 4
movq mm6, mm2                ; do the same on the 2nd input
pand mm6, mmword ptr mask3
paddusb mm5, mm6
```

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

```
pandn mm2, mmword ptr mask3
psrlq mm2, 2
paddusb mm1, mm2 ; accumulate the 'main' result
movq mm6, mm3 ; do the same on the 3rd input
pand mm6, mmword ptr mask3
paddusb mm5, mm6
pandn mm3, mmword ptr mask3
psrlq mm3, 2
paddusb mm1, mm3
movq mm6, mm4 ; do the same on the 4th input
pand mm6, mmword ptr mask3
paddusb mm5, mm6
pandn mm4, mmword ptr mask3
psrlq mm4, 2
paddusb mm1, mm4
psrlq mm5, 2 ;divide the rounding bits by 4 and add them in
paddusb mm1, mm5
```

5.2. Speed

The static speed of the code segments are given in the table below. The segments are named according to the appropriate MB type and half-pel case:

P-type, Y component, Full/FullPYFF

P-type, Y component, Half/FullPYHF

P-type, Y component, Full/HalfPYFH

P-type, Y component, Half/HalfPYHH

B-type, Y component, Full/FullBYFF

The table is calculated under the assumption that the Pentium processor cache line is 32 bytes and that the data stream is uniformly adjusted so that eight L1 misses are expected on average.

	Source Aligned	Source Misaligned	Average 1
PYFF	84	129	123
PYHF	230	298	299 2
PYFH	152	223	214
PYHH	272	362	418 2
BYFF	180	277/372	339 3

Notes:

1. Assuming uniform distribution of source pointer.
2. Code for "esi mod 8 =7" is using the aligned code version with misaligned accesses. This code can be further optimized by writing special code segments for these cases.
3. BYFF actually has two misaligned cases: one in which the source is misaligned and another in which both sources are misaligned. The average is calculated assuming uniform distribution of both pointers. Again, the example code can be further optimized by writing special code segments

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

for these cases. One word of warning, if the code size increases too much in size, there maybe a negative performance effect because of L1 cache misses.

Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback

March 1996

6.0. REFERENCES

MPEG standard ISO/IEC DIS 11172

Berkeley's mpeg_play codecan be downloaded from Berkeley University's FTP site. For example:

`www-plateau.cs.berkeley.edu/mpeg/mpeg_play.html`

`ftp://ftp.cs.berkeley.edu/pub/pix/mm/play`