Using MMX™ Instructions to Perform Simple Vector Operations

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.
CONTENTS

1.0. INTRODUCTION
2.0. MMX_AND FUNCTION
   2.1. Core of mmx_and
   2.2. Alignment Code
   2.3. Other Parts of mmx_and
3.0. MMX_AND PERFORMANCE GAINS
   3.1. Scalar Performance for mmx_and
   3.2. MMX Code Performance for mmx_and
4.0. MMX_ADD FUNCTION
   4.1. Core of mmx_add
   4.2. Other Parts of mmx_add
5.0. MMX_ADD PERFORMANCE GAINS
   5.1. Scalar Performance for mmx_add
   5.2. MMX Code Performance for mmx_add
6.0. MMX_AND FUNCTION: CODE LISTING
7.0. MMX_ADD FUNCTION: CODE LISTING
1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents two examples which demonstrate how to use these instructions to perform basic arithmetic and logic operations on vectors of numbers. Specifically, the \texttt{mmx\_and} and \texttt{mx\_madd} functions demonstrate the use of MMX technology \texttt{PADDUSB} and \texttt{PAND} instructions. MMX technology speeds up such operations significantly as compared to traditional IA (scalar) code, because the new instructions permit basic calculations with 64-bit operands (eight bytes, four words, or two doublewords) in a single clock. The performance gain is also due to the fact that MMX instructions (unlike scalar instructions) can execute operations with a memory operand in one clock.

2.0. MMX\_AND FUNCTION

The \texttt{mmx\_and} function receives as input two byte vectors, performs an AND operation between corresponding elements of the first and second vectors, and stores the result in the second vector. Even though the code example, listed at the end of this note, uses byte vectors, the code could be easily changed to accommodate elements of different lengths. The core loop will not change since these operations are performed bitwise.

The inputs of the \texttt{mmx\_and} function are two pointers to the arrays, and an integer denoting their length (in bytes). Note that this function was designed as a general-purpose library routine, which must be able to handle vectors of any length and alignment. Special-purpose routines written for particular cases will be smaller and probably faster.

2.1. Core of \texttt{mmx\_and}

The core of the \texttt{mmx\_and} function is listed in Example 1.

\textit{Example 1. Core of \texttt{mmx\_and}}

\begin{verbatim}
loop_64:    ; operating on 64 bytes at a time
    1 MOVQ mm0, [edi+0] ; load dst[0-7]
    2 MOVQ mm1, [edi+8] ; load dst[8-15]
    3 MOVQ mm2, [edi+16] ; load dst[16-23]
    4 MOVQ mm3, [edi+24] ; load dst[24-31]
    5 MOVQ mm4, [edi+32] ; load dst[32-39]
    6 MOVQ mm5, [edi+40] ; load dst[40-47]
    7 MOVQ mm6, [edi+48] ; load dst[48-55]
    8 MOVQ mm7, [edi+56] ; load dst[56-63]
    9 PAND mm0, [esi+ 0] ; AND with src[0-7]
   10 PAND mm1, [esi+ 8] ; AND with src[8-15]
   11 PAND mm2, [esi+16] ; AND with src[16-23]
   12 PAND mm3, [esi+24] ; AND with src[24-31]
   13 PAND mm4, [esi+32] ; AND with src[32-39]
   14 PAND mm5, [esi+40] ; AND with src[40-47]
   15 PAND mm6, [esi+48] ; AND with src[48-55]
   16 PAND mm7, [esi+56] ; AND with src[56-63]
   17 MOVQ [edi+ 0], mm0 ; store dst[0-7]
   18 MOVQ [edi+ 8], mm1 ; store dst[8-15]
   19 MOVQ [edi+16], mm2 ; store dst[16-23]
   20 MOVQ [edi+24], mm3 ; store dst[24-31]
   21 MOVQ [edi+32], mm4 ; store dst[32-39]
\end{verbatim}
Using MMX™ Instructions to Perform Simple Vector Operations

March 1996

This core runs on 64 bytes at a time. It is reached only if the vectors contain at least 64 bytes, which start at an address aligned (for the destination vector) to 8 bytes. (Bytes before and after these are operated on by different loops). Note the use of MMX registers—all 8 of them are in use to process 64 bytes at a time. At the start of the loop, ESI contains a pointer to the source array, EDI points to the destination array and EBX contains the number of bytes remaining divided by 64.

Since the loop contains all the MMX code in the function, the EMMS instruction can be placed at the end of the loop. The EMMS instruction is required before exiting from the function since it is a library function that may be called from floating-point code.

2.2. Alignment Code

Since mmx_and can operate on vectors which are not aligned to eight bytes, and misaligned accesses carry a heavy penalty if they appear in the core loop, it is necessary to avoid them. Misaligned accesses cannot be avoided on both the source and destination vectors, as they may be aligned differently. Since each element of the destination vector is accessed twice, and the source vector only once, misaligned accesses are avoided only for the destination vector. This is done in a separate loop which first operates on the bytes before the first 8-byte boundary (of the destination vector) one-by-one. This alignment code appears in Example 2.

**Example 2. Alignment code**

```
1   AND    ebx, 7  ; check dst 8-byte alignment
2   JZ      aligned  ; if aligned skip next section
3   NEG    ebx  ; calculate # of non-aligned bytes
4   ADD    ebx, 8  ; at beginning: 8 - (dst % 8)
5   SUB    ecx, ebx ; n -= (# of non-aligned bytes)
non_al_start:
6   MOV    al, [esi] ; loop to AND non-aligned bytes at
7   INC    esi  ; the start of the vector
8   AND    [edi], al
9   INC    edi
10  DEC    ebx
11  JNZ    non_al_start
aligned:   ; here pointers are 8-byte aligned
```

Before the alignment code is executed, the destination vector pointer is copied to EBX. ECX contains the number of bytes to be operated on. This code is an example of avoiding misaligned accesses in the MMX technology portion of the function.
2.3. Other Parts of mmx_and

Another interesting part of the mmx_and code is the loop which operates on 16 bytes at a time (loop_16 in the code listing). The loop is used for vectors which are less than 64 bytes long, or for the bytes remaining after the last 64-byte chunk. Note that it is not MMX code, thus MMX code will not be executed for function calls with vectors shorter than 64 bytes. This is an important consideration since the function might be called from floating point code, and the small savings for short vectors (0 to 6 clocks per function call, see Section 3.2) is outweighed by the overhead of changing from floating point to MMX code and back. If it is known that the function will not be called from floating point code, there is no reason not to convert this loop to MMX instructions as well.
3.0. MMX_AND PERFORMANCE GAINS

This section describes the performance improvement compared with traditional IA scalar code. There is a 1.3X improvement in the core loop which will be close to the overall performance improvement for vectors much longer than 64 bytes, assuming all data is in the cache, and that the source and destination vectors have the same 8-byte alignment. Performance gains are reduced if there are many cache misses or misaligned accesses.

3.1. Scalar Performance for mmx_and

The scalar version of the core loop is the 16-byte loop unrolled four times. Assuming no misaligned accesses and no cache misses, it takes two clocks for every four bytes, plus a loop overhead of two clocks, for a total of 34 clocks per 64-byte iteration.

3.2. MMX Code Performance for mmx_and

Assuming there are no misaligned accesses and no cache misses, the MMX code core loop takes 26 clocks to execute. This is 1.3 times the performance of the scalar loop. MMX technology enables 2 times peak throughput on AND operations (128 bits/clock vs. 64 for scalar code). The performance gain is significantly less than 2 times because this loop is memory-access limited: its execution speed is bound not by the ALU operations, but by the loads and stores. (Remember that scalar IA instructions can access the same amount of memory per clock as MMX instructions-64 bits-by pairing two 32-bit accesses). The reason that any gain is seen is the fact that MMX instructions (unlike scalar instructions) can execute operations with a memory operand in one clock. Thus, scalar IA code needs four clocks to load 64 bits twice, do a 64-bit AND, and store 64 bits, while MMX instructions can do the same in three clocks.

Note that the MMX code has many available pairing slots (see next section), which can be used by any MMX instructions which do not access memory. Thus, additional operations could be performed on the operands or the result without increasing execution time. For example, each result could be shifted before storing. In this case, the performance gain compared to the scalar code would be higher.

Instruction Scheduling, Pairing and Stalls

The mmx_and core is an example which demonstrates pairing limitations. The scheduling of the instructions is simple and straightforward, and the MMX instructions are not paired at all. The reason is that all the MMX instructions access memory, so they cannot pair with each other no matter they are rescheduled. They also cannot pair with scalar instructions. This is an example of the rule:

The rule shows the number of clocks that cannot be further reduced by instruction scheduling. This is another way of looking at the memory-access limitation discussed in the previous section.

\[
\text{Clocks} \geq (\text{MMX_memory_accesses} + \frac{\text{non-MMX_instructions}}{2})
\]
4.0. MMX_ADD FUNCTION

The **mmx_add** function is similar to **mmx_and** in structure. It receives two byte vectors, and performs an 8-bit addition (with unsigned saturation) between each element of the first vector and the corresponding element of the second, storing the result in the second vector. Again, although the code example uses byte vectors, the code could be easily changed to perform addition on word vectors, for which MMX technology also supports unsigned saturation.

Though the method used in **mmx_add** is basically similar to **mmx_and**, there are several differences, particularly in the granularity of the various loops. The **mmx_add** core loop processes 32 bytes per iteration, and there is a scalar loop which operates on two bytes at a time, unlike 16 for **mmx_and** (the loops are easily unrolled if needed). The loops which operate on bytes one-by-one at the start and end of the vectors are also slightly different (see code listing at the end of the note).

The inputs of the **mmx_add** function are two pointers to the arrays, and an integer denoting their length (in bytes). This function was designed as a general-purpose library routine, which must be able to handle vectors of any length and alignment. Special-purpose routines written for particular cases will be smaller and probably faster.

4.1. Core of mmx_add

The core of the **mmx_add** function is listed in Example 3.

```
Example 3. Core of mmx_add

loop_32:       ; operating on 32 bytes at a time
    1  MOVQ   mm0, [edi+0]  ; load dst[0-7]
    2  MOVQ   mm1, [edi+8]  ; load dst[8-15]
    3  MOVQ   mm2, [edi+16] ; load dst[16-23]
    4  MOVQ   mm3, [edi+24] ; load dst[24-31]
    5  PADDUSB mm0, [esi+ 0] ; ADD with src[0-7]
    6  PADDUSB mm1, [esi+ 8] ; ADD with src[8-15]
    7  PADDUSB mm2, [esi+16] ; ADD with src[16-23]
    8  PADDUSB mm3, [esi+24] ; ADD with src[24-31]
    9  MOVQ   [edi+ 0], mm0  ; store dst[0-7]
   10  MOVQ   [edi+ 8], mm1  ; store dst[8-15]
   11  MOVQ   [edi+16], mm2 ; store dst[16-23]
   12  MOVQ   [edi+24], mm3 ; store dst[24-31]
   13  ADD    edi, 32        ; update dst pointer
   14  ADD    esi, 32        ; update src pointer
   15  SUB    eax, 32
   16  JG     loop_32        ; there is no more multimedia code

This core runs on 32 bytes at a time (if desired, it can easily be changed to operate on 64 bytes per iteration, like the core loop of **mmx_and**). It is reached only if the vectors contain at least 32 bytes, which start at an address aligned (for the destination vector) to eight bytes. (Bytes before and after these are operated on by different loops). At the start of the loop, **ESI** contains a pointer into the source array, **EDI** points into the destination array, and **EAX** contains the number of bytes remaining.
```
Since the loop contains all the MMX code in the function, the EMMS instruction can be placed at the end of the loop. The EMMS instruction is required before exiting the function since it is a library function that may be called from FP code.

### 4.2. Other Parts of mmx_add

Similarly to mmx_and, a scalar loop processes any bytes which remain at the end of the vectors (loop_2 in the code listing). It processes two bytes at a time—this is the largest number of 8-bit additions which can be effectively paralleled in scalar IA code. The alignment code which operates on misaligned bytes at the start of the vectors is similar, though not identical, to the alignment code in mmx_and. (seven lines before non_al_start in the code listing).
5.0. MMX_ADD PERFORMANCE GAINS

This section describes the performance improvement as compared with traditional IA scalar code. There is an improvement in the core loop ranging from 6 times to 7-18 times, depending on whether the input data causes many adds to saturate. This gain will be close to the overall performance improvement for vectors much longer than 32 bytes, assuming all data is in the cache, and that the source and destination vectors have the same 8-byte alignment. Performance gains are reduced if there are many cache misses or misaligned accesses.

5.1. Scalar Performance for mmx_add

A scalar version of the core loop would be similar to the 2-byte loop unrolled 16 times. Assuming no misaligned accesses, no cache misses and no addition saturations, execution of each iteration would take 84 clocks. Each of the 32 additions that saturates adds one or six clocks, depending on whether the branch was correctly predicted. This totals 84 to 100-250 clocks, depending on the amount of saturations and branch mispredictions.

5.2. MMX Code Performance for mmx_add

Assuming no misaligned accesses and no cache misses, the MMX code core loop takes 14 clocks to execute. This represents a speedup of 6 times to about 7-18 times as compared to the scalar IA code, depending on the amount of saturations and branch mispredictions. The additional gain above 6 times is due to the MMX technology hardware support for saturation. MMX technology has a peak rate for 8-bit ADD operations which is 8 times that of scalar IA code (16 8-bit adds/clock vs. two). However, this code is memory-access-limited, like the core loop of mmx_and, so the base speedup is lower than 8 times. Furthermore, the fact that MMX instructions can execute an operation with a memory operand in one clock improves speedup.

Similarly to mmx_and, the core loop of mmx_add has many available pairing slots, which can be used by any MMX instruction which does not access memory. Thus additional operations could be performed on the operands or the result without increasing execution time. For example, each result could be shifted before storing. In this case the speedup would be higher when compared to the IA code, since adding more operations to the scalar code would increase its execution time.

The scheduling and pairing considerations for the core loop of mmx_add are similar to those for mmx_and.
6.0. MMX_AND FUNCTION: CODE LISTING

```asm
.486P
.model FLAT
PUBLIC _mmx_and
_DATA SEGMENT
_DATA ENDS
_TEXT SEGMENT
_mmx_and       PROC NEAR
src EQU 16[esp]
dst EQU 20[esp]
n EQU 24[esp]
push ebx
push esi
push edi
mov esi, src  ; pointer to src vector
mov ecx, n   ; size of vectors (in bytes)
mov edi, dst  ; pointer to dst vector
cmp ecx, 0   ; compare n to 0
mov ebx, edi  ; copy pointer for alignment calc.
jle Exit   ; exit if n <= 0
cmp ecx, 8   ; compare n to 8
jl end_bytes  ; if n < 8, jump to end loop
and ebx, 7   ; check dst 8-byte alignment
jz aligned   ; if aligned skip non-aligned section
neg ebx   ; calculate number of non-aligned bytes
add ebx, 8   ; at beginning of dst: 8 - (dst % 8)
sub ecx, ebx   ; n -= (# of non-aligned bytes)

non_al_start:
    mov al, [esi]  ; loop to AND the non-aligned (dst) bytes at
    inc esi   ; the start of the vectors
    ;
    and [edi], al
    inc edi
    ;
    dec ebx
    jnz non_al_start

aligned:    ; destination pointer is now 8-bytes aligned
    mov ebx, ecx
    ; copy n
    shr ebx, 6   ; ebx = n / 64
    jz after_loop_64  ; if n < 64 skip 64-byte loop

loop_64:    ; operating on 64 bytes at a time
    movq mm0, [edi+ 0]  ; load dst[0-7]
movq mm1, [edi+ 8]  ; load dst[8-15]
movq mm2, [edi+16]  ; load dst[16-23]
movq mm3, [edi+24]  ; load dst[24-31]
movq mm4, [edi+32]  ; load dst[32-39]
movq mm5, [edi+40]  ; load dst[40-47]
movq mm6, [edi+48]  ; load dst[48-55]
movq mm7, [edi+56]  ; load dst[56-63]
pand mm0, [esi+ 0]  ; AND with src[0-7]
pand mm1, [esi+ 8]  ; AND with src[8-15]
pand mm2, [esi+16]  ; AND with src[16-23]
pand mm3, [esi+24]  ; AND with src[24-31]
pand mm4, [esi+32]  ; AND with src[32-39]
 Exit:
    ret
end_bytes:
    ret
```

March 1996
Using MMX™ Instructions to Perform Simple Vector Operations

March 1996

```
pand  mm5, [esi+40] ; AND with src[40-47]
pand  mm6, [esi+48] ; AND with src[48-55]
pand  mm7, [esi+56] ; AND with src[56-63]
movq  [edi+ 0], mm0 ; store dst[0-7]
movq  [edi+ 8], mm1 ; store dst[8-15]
movq  [edi+16], mm2 ; store dst[16-23]
movq  [edi+24], mm3 ; store dst[24-31]
movq  [edi+32], mm4 ; store dst[32-39]
movq  [edi+40], mm5 ; store dst[40-47]
movq  [edi+48], mm6 ; store dst[48-55]
movq  [edi+56], mm7 ; store dst[56-63]
add   edi, 64  ; update dst pointer
add   esi, 64  ; update src pointer
sub   ebx, 1
jg    loop_64
emms

after_loop_64:
    and  ecx, 63   ; if (n % 64) == 0, no more bytes remain
    jz     Exit
    mov   ebx, ecx  ; copy n
    shr   ebx, 4   ; ebx = n / 16
    jz     after_loop_16

loop_16:
    mov   eax, [esi]
    mov   edx, [edi]
    and   eax, edx
    mov   edx, [edi+4]
    mov   [edi], eax
    mov   eax, [esi+4]
    and   eax, edx
    mov   edx, [edi+8]
    mov   [edi+4], eax
    mov   eax, [esi+8]
    and   eax, edx
    mov   edx, [edi+12]
    mov   [edi+4], eax
    mov   eax, [esi+12]
    and   eax, edx
    add   esi, 16
    mov   [edi+12], eax
    add   edi, 16
    sub   ebx, 1
    jg    loop_16

after_loop_16:
    and  ecx, 15
    jz    Exit

end_bytes:   ; AND bytes at end one-by-one
    mov   al, [esi+ecx-1]
    and   [edi+ecx-1], al
    sub   ecx, 1
    jnz   end_bytes

Exit:
    pop   edi
    pop   esi
    pop   ebx```
Using MMX™ Instructions to Perform Simple Vector Operations

March 1996

ret     0
_mmx_and ENDP
_TEXT    ENDS
END
7.0. MMX_ADD FUNCTION: CODE LISTING

```
.model FLAT
PUBLIC    mmx_add
_DATA   SEGMENT
_DATA ENDS
_TEXT   SEGMENT
_mmx_add PROC NEAR

_UCHAR_MAX = 255 ; Max unsigned byte (for saturation)
src EQU [esp+16]
dst EQU [esp+20]
n EQU [esp+24]
push esi
push edi
push ebx
mov edi, dst ; pointer to dst vector
mov esi, src ; pointer to src vector
mov eax, n ; size of vectors (in bytes)
mov ebx, edi
and ebx, 7 ; check dst 8-byte alignment
jz aligned ; if aligned skip non-aligned section
neg ebx ; calculate number of non-aligned bytes
add ebx, 8 ; at beginning of dst: 8 - (dst % 8)
cmp eax, ebx ; compare n to # of non-aligned bytes
jl check_end ; if n is smaller jump to end loop, check n>0
sub eax, ebx ; n -= (# of non-aligned bytes)
non_al_start:
    mov cl, [esi] ; loop to ADD the non-aligned (dst) bytes at
    inc esi ; the start of the vectors. (saturate result)
    ;
mov dl, [edi] ;
    inc edi ;
    add dl, cl ;
jnc dont_saturate0 ; if carry bit not set, do not saturate
    ;
mov dl, UCHAR_MAX ; saturate to max unsigned byte
    ;
dont_saturate0:
    ;
    mov [edi-1], dl ;
    sub ebx, 1 ;
jnz non_al_start ;
aligned: ; destination pointer is now 8-bytes aligned
    sub eax, 32 ; subtract 32 from n
    jl after_loop_32 ; if n was less than 32, skip 32-byte loop
loop_32: ; operating on 32 bytes at a time
    movq mm0, [edi+ 0] ; load dst[0-7]
movq mm1, [edi+ 8] ; load dst[8-15]
movq mm2, [edi+16] ; load dst[16-23]
movq mm3, [edi+24] ; load dst[24-31]
paddusb mm0, [esi+ 0] ; ADD with src[0-7]
paddusb mm1, [esi+ 8] ; ADD with src[8-15]
paddusb mm2, [esi+16] ; ADD with src[16-23]
```

Using MMX™ Instructions to Perform Simple Vector Operations
March 1996

paddusb mm3, [esi+24] ; ADD with src[24-31]
movq [edi+ 0], mm0 ; store dst[0-7]
movq [edi+ 8], mm1 ; store dst[8-15]
movq [edi+16], mm2 ; store dst[16-23]
movq [edi+24], mm3 ; store dst[24-31]
add edi, 32 ; increment dst pointer
add esi, 32 ; increment source pointer
sub eax, 32
jge loop_32 ; there is no more multimedia code
after_loop_32:
add eax, 32 ; add back the 32 we subtracted from n
jle Exit ; if n <= 0 exit
sub eax, 2 ; check if only one byte remains
jl last_byte ; skip over loop_2 if so
loop_2: ; operating on 2 bytes at a time
mov cl, [edi+eax]
mov bl, [esi+eax]
mov ch, [edi+eax+1]
mov dl, [esi+eax+1]
add bl, cl
jnc dont_saturate1 ; if carry bit not set, do not saturate
mov bl, UCHAR_MAX ; saturate to max unsigned byte
dont_saturate1:
add dl, ch
jnc dont_saturate2 ; if carry bit not set, do not saturate
mov dl, UCHAR_MAX ; saturate to max unsigned byte
dont_saturate2:
mov [edi+eax], bl
mov [edi+eax+1], dl
sub eax, 2
jge loop_2
add eax, 1 ; if no bytes remain exit
jl Exit
end_bytes: ; AND bytes at end one-by-one
mov cl, [edi+eax]
mov bl, [esi+eax]
add bl, cl
jnc dont_saturate3 ; if carry bit not set, do not saturate
mov bl, UCHAR_MAX ; saturate to max unsigned byte
dont_saturate3:
mov [edi+eax], bl
check_end:
sub eax, 1
jge end_bytes
jmp Exit
last_byte:
add eax, 1
jmp end_bytes
Exit:
pop edi
pop esi
pop ebx
ret 0

_mmx_add ENDP
_TEXT ENDS