



Using MMX™ Instructions to Get Bits From a Data Stream

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 Get Bits From a Data Stream

March 1996

CONTENTS

1.0. INTRODUCTION

2.0. GETBITS FUNCTION

2.1. Getbits Core

2.2. new64bit Code

2.3. do_refill Code

3.0. PERFORMANCE GAINS

3.1. Scalar Performance

3.2. MMX™ Code Performance

4.0. GETBITS FUNCTION: CODE LISTING

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, the `getbits` function presented here illustrates how to use the new MMX™ instructions (`PSRLQ` and `PSLLQ`) to manipulate a data stream. The performance improvement relative to traditional IA code can be attributed primarily to the much faster shift instructions. Whereas the IA shift instruction (`SHIFT`) takes four cycles on a Pentium® processor, the MMX instruction (`PSHIFT`) takes only one cycle. The performance gain is also due to the fact that the MMX instructions operate on 64-bit values instead of the 32-bit values operated on by the scalar shift instructions.

2.0. GETBITS FUNCTION

In applications such as the Moving Pictures Expert Group (MPEG) applications, the data is organized in a bit stream in big-endian order. The user requires a variable number of bits from this bitstream. The `getbits` function performs these tasks to access the requested data:

1. The function gets the user-specified number of bits from the multimedia register, `MM0`, and returns them in the integer register, `EAX`.
2. When all the bits in `MM0` have been used, `getbits` jumps to the `new64bit` code. This code refreshes `MM0`, filling it with bits from a buffer of finite length. `getbits` then returns the requested bits in the `eax` register.
3. When there are no more bits in the buffer, `getbits` jumps to the `do_refill` code, which places a finite number of bits from the input stream into the buffer.

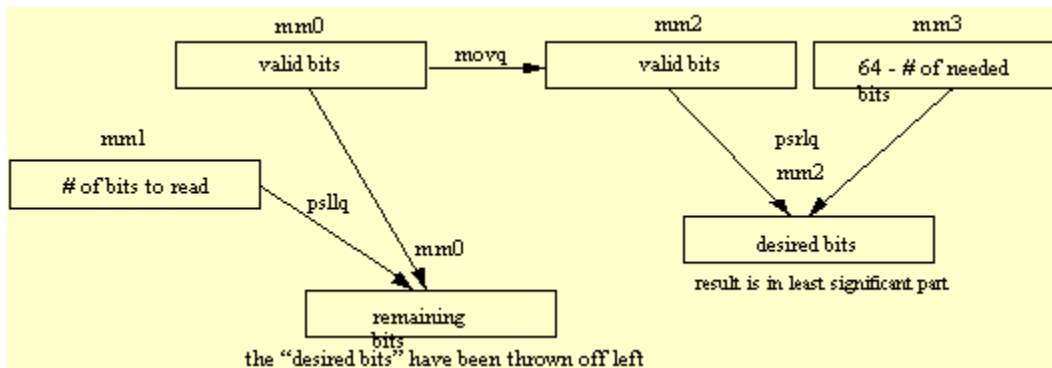
2.1. Getbits Core

Figure 1 illustrates the basic MMX code flow needed for the `getbits` core. Basically, `MM2` (a copy of `MM0`) shifts the desired bits all the way to the least significant part of the register. Register `MM0` is updated so that these returned bits are removed.

The core of the `getbits` function is listed in Example 1. Recall that the multimedia register, `MM0`, contains the bits that will be retrieved by `getbits`.

The first three lines determine how many bits the user requests and if there are enough valid bits in the `MM0` register. Assuming that `MM0` contains enough valid bits for this call of `getbits`, the branch in line 4 is not taken.

Figure 1. Getbits Core Diagram



Example 1. Getbits Core

```

getbits:
1  MOV     eax, DWORD PTR bit_count ; Number of valid bits in MM0
2  MOV     ecx, 4[ESP]             ; Parameter. How many bits to read
3  SUB     eax, ecx                ; Do we have enough bits in MM0
4  JL     new64bit                 ; If not get new 64 bits
5  movd   MM3,64_minus_index[ecx*4]; MM3 = 64 - number of needed bits
6  movq   MM2,MM0                 ; make a copy of the bits
7  movd   MM1,ecx                 ; number of bits to read
8  psrlq  MM2,MM3                 ; MM2 now has valid bitstream in
                                ; least significant part
    
```

Using MMX™ Instructions to Get Bits From a Data Stream

March 1996

```
9     mov     BYTE PTR bit_count,al    ; Update number of valid bits
10    movd   eax,MM2                  ; move the result into eax
11    psllq  MM0,MM1                  ; throw away those bits
12    ret
```

The big-endian to little-endian conversion is already performed. See Section 2.2. `new64bit` Code. The relevant bits are currently in the most significant part of `MM0`. These bits are copied to `MM2` (line 6). The bits in `MM2` are shifted to the right so that they are now in the least significant part of the register (lines 5 and 8). The bits can then be loaded to the `EAX` register (line 10). The bits that were just loaded to `EAX` are then discarded at the most significant edge of `MM0` (shift to left in lines 7 and 11). Line 9 updates the number of valid bits in `MM0`.

The read from a 64-bit MMX register takes only eight cycles, including the two cycles for the `RET` instruction. If the application uses this core piece of the `getbits` function as a macro, the two-cycle overhead for the return is saved. Of course, this means an extra 42 bytes of code size each time this macro is called.

In addition, if another register can be spared, then the number of requested bits can be passed to the macro in the `ECX` register, or another similar register. The code may then take as few as four cycles (another instruction or a pair of instructions could execute during the free cycle). The restructured code is listed in Example 2.

Example 2. Restructured getbits Core

```
getbits:
1     sub     edi,ecx                  ; Are there enough bits in MM0
2     jnl    continue                ; If there are, then continue
3     call   new64bit                 ; If not, get new 64 bits
4     jmp    out                       ; finished
continue:
5     movd   MM3,64_minus_index[ecx*4] ;MM3 = 64 - number of requested bits
6     movq   MM2,MM0                  ; make a copy of the bits
7     movd   mm1,ecx                  ; number of bits to read
8     psrlq  MM2,MM3                  ; MM2 has valid bitstream in least significant part
                                           ;(free cycle)
                                           ;(free cycle)
10    movd   eax,MM2                  ; move the result into eax
11    psllq  MM0,MM1                  ; throw away those bits
out:
```

2.2. `new64bit` Code

The `new64bit` code is performed when there are not enough bits available in `MM0`. This code (as illustrated in Figure 2) calculates the number of requested bits less the number of bits remaining in `MM0`. A temporary copy of the bits remaining in `MM0` is made. Then, another 64 bits are obtained from the bitstream (in two 32-bit halves that are converted to little endian at the start of the `refill` code). The remaining bits are prepended to these bits. After the requested bits are returned in the `EAX` register, `MM0` contains the remaining part of the new 64 bits (the new bits are appended to the previously remaining bits).

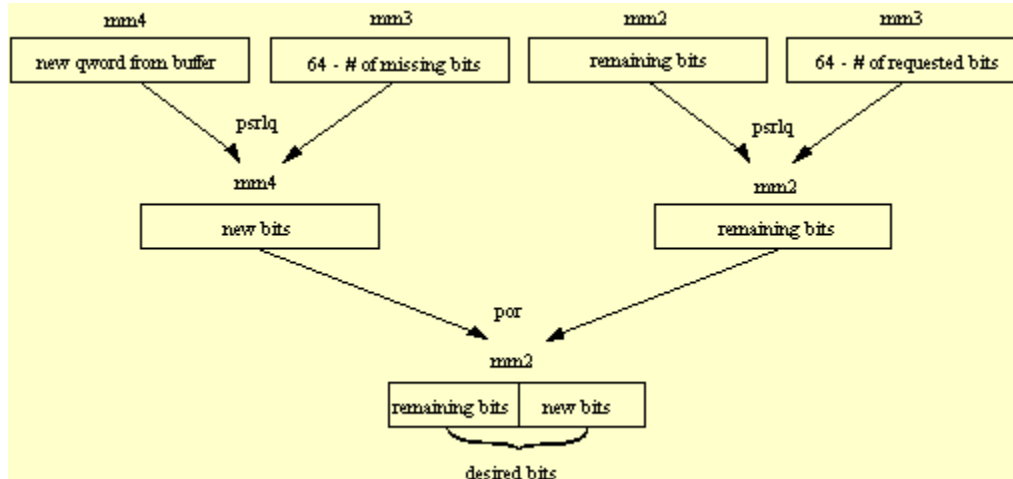
The code is listed in Example 3.

Using MMX™ Instructions to Get Bits From a Data Stream

March 1996

Actual statistics from a sample MPEG1 audio application shows that an average of 4.86 bits are requested each time `getbits` is called. This means that an average of once every 13.17 calls to `getbits`, 64 more bits are read from the buffer into `MM0`.

Figure 2: new64bits Core Diagram



Example 3. new64bit Code

```

group), (negative)
1      add     eax,64
2      movd   mm3,_64_minus_index[ecx*4]
3      psrlq  mm2,mm3
4      movd   mm3,eax
5      movq   mm0,mm4
6      psrlq  mm4,mm3
7      mov    _bit_count,eax
8      por    mm2,mm4
9      movd   mm1,_64_minus_index[4*eax]
10     movd   eax,mm2
11     psllq  mm0,mm1
;mm2 has remaining bits from old group
;mm4 has new 64 bits
;eax has -(number of bits we missed in old
;ecx has number of requested bits
; eax = 64- # of bits we missed in old
; group
; MM3 = 64 - number of requested bits
; mm2 has remaining bits in least
; significant part with room for new
; bits to right of it
; mm3=64- # of bits we missed in old
; group
; save the new word in mm0 for next time
; mm4 = new bits we now need in least
; significant part
; Save bit count for next time
; combine remaining bits with the bits
; from new word
; # of bits we missed in old group
; return bits in eax
; remove the bits we just read from mm0

```

2.3. do_refill Code

When the 64-bit aligned input buffer is empty, the `getbits` function performs the `do_refill` code. The `do_refill` code resets the values of `buf_pointer` (pointer to input buffer) and the `EndBuf` (pointer to end of buffer). If the application assumes that the buffer is never empty (for example, perhaps the buffer is filled at the end of every frame) the two-cycle overhead of testing for end of buffer can be saved.

The `do_refill` code reads new data from the input stream every 64 bits. The current 64 bits are stored in `MM0`. Therefore, other functions should not modify the contents of `MM0`.

3.0. PERFORMANCE GAINS

This section details the performance improvement as compared with traditional scalar code. There is approximately a 2X performance gain for the MMX optimized code version for typical MPEG Audio code.

3.1. Scalar Performance

The scalar version of the `getbits` function executes in 18 cycles (if there are enough bits in the 32-bit register). In case of a jump to the `new32bit` code (with a mispredicted branch it must be called more often since it retrieves only 32 bits), the code takes 38 cycles to execute, plus an additional five cycles (for the misprediction) to read another 32 bits.

Assuming 4.86 bits per read (see Section 2.2), the 32-bit register is refilled approximately every 6.58 reads. Total `getbits` execution time (on average) therefore is:

$$(18 * (6.58-1) + (38+5)) / 6.58 = 21.8 \text{ cycles}$$

3.2. MMX™ Code Performance

The MMX optimized code version of the `getbits` function executes in eight cycles (if there are enough bits in `MM0`). The speedup can be attributed to several factors:

The MMX code uses 64-bit registers, rather than the 32-bit registers used by scalar code.

By using a 64-bit read, the cost of the mispredicted branch is reduced by half, compared to the scalar code with a 32-bit read.

In case of a jump to the `new64bit` code, the `getbits` function takes 19 cycles to execute, plus an additional five cycles (for the misprediction) to read another 64-bit value.

Again, assuming 4.86 bits per read (see Section 2.2), the 64-bit `MM0` register is refilled approximately every 13.17 `getbits` calls. Therefore, on average, the total execution time is:

$$(8 * (13.17-1) + (19+5)) / 13.17 = 9.2 \text{ cycles}$$

It takes 9.2 cycles to finish the MMX technology version of the `getbits` call. The performance gain is almost 2.4X over that of typical MPEG Audio code.

With the Pentium® Pro processor, the misprediction penalty is much higher. The performance gain relative to scalar code will therefore be even greater.

Note that this analysis assumes 100 percent misprediction on the branch.

Using MMX™ Instructions to Get Bits From a Data Stream

March 1996

4.0. GETBITS FUNCTION: CODE LISTING

```
.486P

ASSUME ds:FLAT, cs:FLAT, ss:FLAT
EXTRN    refill_buffer:PROC
DATA SEGMENT PARA    PUBLIC USE32 'DATA'
    ALIGN    16
64_minus_index    dd
64,63,62,61,60,59,58,57,56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,34
    dd
33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1
EXTRN    _buf_pointer:DWORD
EXTRN    _end_buf:DWORD
EXTRN    _bit_count:DWORD
_DATA ENDS
_TEXT SEGMENT DWORD PUBLIC USE32 'CODE'
PUBLIC    _getbits
_getbits:
    mov     eax, DWORD PTR _bit_count ; Number of valid bits in MM0
    mov     ecx, 4[ESP]                ; Parameter. How many bits should we
                                        ; read.
    sub     eax,ecx                    ; Do we have enough bits in MM0
    jl     new64bit                    ; If not get new 64 bits
    movd   MM3,_64_minus_index[ecx*4] ; MM3 = 64 - number of needed bits.
    movq   MM2,MM0

    movd   mm1,ecx                    ; # of bits to read
    psrlq  MM2,MM3                    ; MM2 now has valid bitstream in least
                                        ; significant part
    mov    BYTE PTR _bit_count,al     ; Update number of valid bits.
    movd   eax,MM2                    ; move the result into eax
    psllq  MM0,MM1                    ; throw away those bits

    ret
new64bit:
    movd   MM3,_64_minus_index[ecx*4] ;MM3 = 64 - number of requested bits
                                        ;(for shifting)
    movq   MM2,MM0                    ; copy left over bits
    mov    edx,DWORD PTR _buf_pointer ;pointer to bitstream
    mov    ecx,_end_buf                ;read pointer to end of buffer
    add    edx,8                       ;update the pointer
    add    eax,64                       ;eax = 64- # of bits we missed in old
                                        ;group
    cmp    edx,ecx                     ;do we have another qword to read
    mov    DWORD PTR _buf_pointer,edx  ;save new value
    mov    ecx,[edx-8]                  ;read next qword (dword here)
    mov    edx,[edx-4]                  ;(dword here)
    jge    _do_refill                   ;do_refill
refill:
    ; now convert from bigendian to little and
    ; but make use of left over bits (MM2) before using these
    bswap  edx                          ;swapping the first 32 bit
    bswap  ecx                          ;swapping the second 32 bit
    movd   mm4,ecx                       ;second 32 bit in mm4
    psrlq  mm2,mm3                       ;mm2 has remaining bits in least
                                        ;significant part with room for new
                                        ;bits to right of it
    movd   mm4,mm1                       ;move first 32 bit
    psllq  mm4,32                         ;shiftsecond 32 bit to upper part of
                                        ;register
    movd   mm3,eax                       ;mm3 gets the shift counter
```

Using MMX™ Instructions to Get Bits From a Data Stream

March 1996

```
    por    mm4,mm1           ;combine the 64 swapped data into mm4
    movq   mm0,mm4          ;save new word in mm0 for next time
    psrlq  mm4,mm3          ;mm4 = new bits we now need in least
                                ;significant part
    mov    bit_count,eax    ;Save bit count for next time
    por    mm2,mm4          ;combine remaining bits with the bits
                                ;from new word
    movd   mm1, _64_minus_index[4*eax] ;# of bits we missed in old group
    movd   eax,mm2          ;return bits in eax
    psllq  mm0,mm1          ;remove the bits we just
                                ;read from mm0
    RET
do_refill:
    PUSH   EAX
    PUSH   EDX
    PUSH   ECX
    CALL   _refill_buffer
    POP    ECX
    POP    EDX
    POP    EAX
    JMP    refill
_TEXT ENDS
END
```