Using MMX™ Instructions to Compute a 16-Bit Vector

Information for Developers and ISVs

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

March 1996

CONTENTS

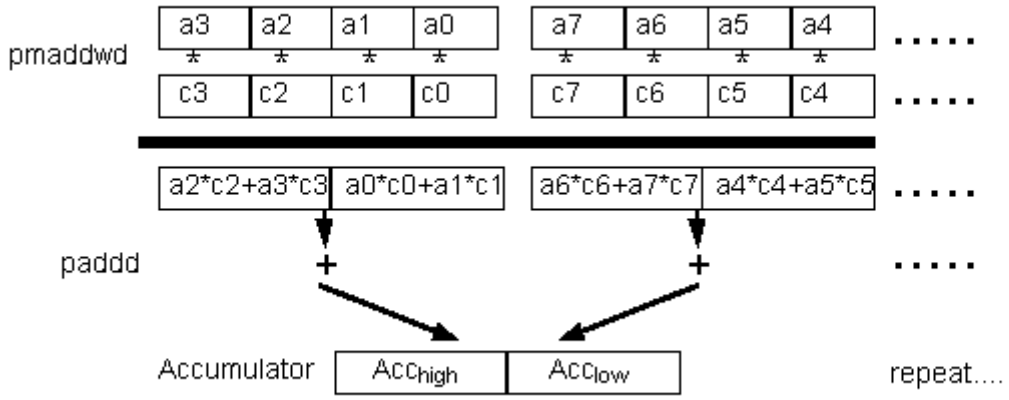
1.0. INTRODUCTION

Calculating the dot-product of two vectors requires executing a large number of multiply-accumulate operations. This application note shows how to use the MMX technology PMADDWD instruction to significantly speed up 16-bit vector dot-product calculation. The PMADDWD instruction multiplies four pairs of 16-bit numbers and produces partial sums of the results - and can do so once per clock (with a three-clock latency). A throughput of up to two 16-bit multiply-accumulates per clock can be achieved if the instructions are correctly scheduled. The results are accumulated with 32-bit precision.

2.0. VDPMMX16 FUNCTION

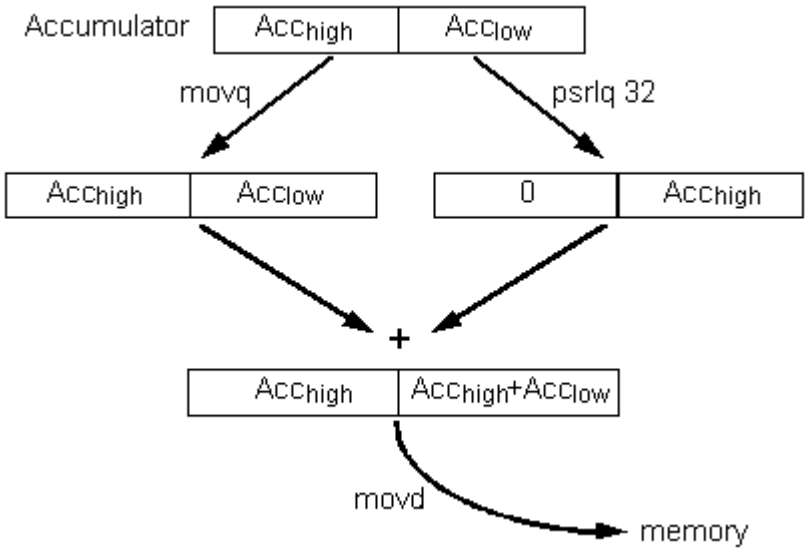
The vector dot-product operation is an basic building block of many common linear algebra and DSP operations, such as matrix multiplication and FIR filters. The VDPMMX16 function, illustrated in Figures 1 and 2, is an MMX technology implementation of the vector dot-product operation. VDPMMX16 performs a dot-product on two 16-bit vectors, calculating the result with 32-bit precision.

Figure 1. Vector Dot-Product in MMX™ Technology - Part I



The MMX technology PMADDWD instruction is used on four elements from each vector. The results are two 32-bit numbers; the sum of the first two products and the sum of the second two products. These are accumulated into an MMX register by using PADDD. This operation is repeated on the next four elements until all the elements have been multiplied. This is the basic loop of the MMX technology vector dot-product implementation - it can be unrolled to achieve better performance.

Figure 2. Vector Dot-Product Using MMX™ Technology - Part II



At this point the accumulator register contains two (doubleword) partial sums. The actual result of the vector dot-product is the sum of these partial sums. That is, the low and high doublewords must be added together. To do this:

Copy the accumulator register.

Shift the duplicated accumulator register 32 bits to the right, so that the doubleword which was in the high half of the accumulator is in the low half of the duplicated accumulator.

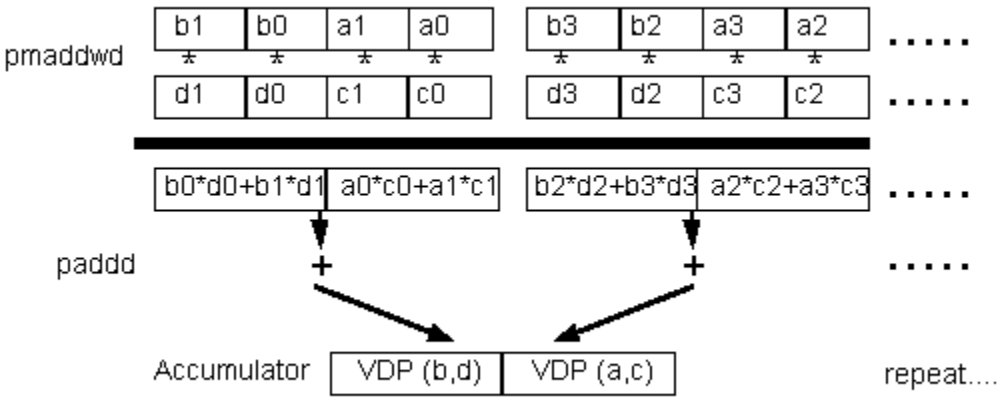
Sum the registers.

Now the desired result is in the low half of the register and it can be written to memory with one MOVD instruction.

2.1. Alternate Method

If the routine can calculate two vector dot-products each time it is called and the data can be interleaved ahead of time without affecting performance, performance can be improved. This is because the accumulation of the partial sums at the end of the loop can be avoided in this case, thereby saving three clocks at the end of the loop (see Figure 3). If the vectors are relatively short, this may be significant.

Figure 3. Alternate Method

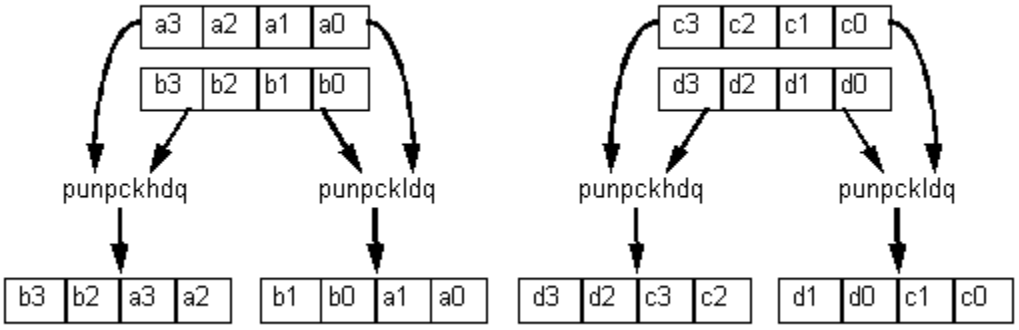


With the input data interleaved, the main loop calculates two separate vector dot-products instead of calculating two partial sums of one vector dot-product. Though this method is quicker in specific cases (short vectors, dot-products can be calculated two at a time, vectors can be interleaved ahead of time), in this document the VDPMMX16 function shows the more general case, and will use the first method illustrated in Figures 1 and 2.

Interleaving the Data

The two sets of vectors can be interleaved using the PUNPCKLDQ and PUNPCKHDQ instructions (see Figure 4). This can be done without affecting performance if the routine which creates the input data has a few available pairing slots, or the same data is created once and used many times.

Figure 4. Interleaving Two Vectors



2.1. vdpmmx16 Core

The core of the VDPMMX16 function is listed in Example 1.

Example 1. vdpmmx16 Core

```
1       PXOR    mm4, mm4        ; prepare for 1st iteration (SW pipelining)
2       PXOR    mm6, mm6        ; prepare for 1st iteration (SW pipelining)
3       PXOR    mm7, mm7        ; initialize accumulator
        ;.align 16
```

```
vdp16:
4       MOVQ    mm0,0[eax]      ; first 4 source1 elements
5       PADDD   mm7, mm4        ; accumulate (from prev. iteration)
6       MOVQ    mm1,0[ebx]      ; first 4 source2 elements
7       PADDD   mm7, mm6        ; accumulate (from prev. iteration)
8       MOVQ    mm2,8[eax]      ; next 4 source1 elements
9       PMADDWD mm0, mm1
        ; s1[0]*s2[0] + s1[1]*s2[1]::s1[2]*s2[2] + s1[3]*s2[3]
10      MOVQ    mm3,8[ebx]      ; next 4 source2 elements
        ;V-pipe empty - memory can be accessed only in U-pipe
11      MOVQ    mm4,16[eax]     ; next 4 source1 elements
12      PMADDWD mm2, mm3
        ; s1[4]*s2[4] + s1[5]*s2[5]::s1[6]*s2[6] + s1[7]*s2[7]
13      MOVQ    mm5,16[ebx]     ; next 4 source2 elements
14      PADDD   mm7, mm0        ; accumulate
15      MOVQ    mm6,24[eax]     ; last 4 source1 elements
16      PMADDWD mm4, mm5
        ; s1[8]*s2[8] + s1[9]*s2[9]::s1[10]*s2[10] + s1[11]*s2[11]
17      PMADDWD mm6, 24[ebx]
        ; s1[12]*s2[12] + s1[13]*s2[13]::s1[14]*s2[14] + s1[15]*s2[15]
18      PADDD   mm7, mm2        ; accumulate
19      ADD     eax, 32 ; increment source1 index
20      ADD     ebx, 32 ; increment source2 index
21      SUB     ecx, 16
22      JNZ     vdp16
23      PADDD   mm7, mm4        ; accumulate from last iteration
                                ; (SW pipelining)
24      PADDD   mm7, mm6        ; accumulate from last iteration
                                ; (SW pipelining)
```

The EAX register contains a pointer to one source vector, the EBX register to the other source vector and ECX contains the number of elements (words) in the vectors. At the end of this section of code MM7 contains two partial sums which are later added to generate the final result.

Several changes were made from the flow shown in Figures 1 and 2 to improve performance. One is that the core loop was unrolled four times; it operates on 16 elements each iteration, instead of four. This yields more scheduling opportunities and improves pairing, in addition to reducing loop overhead.

Also software pipelining (scheduling together instructions from different operations or loop iterations) has been used to achieve maximum pairing. The instructions PADDD MM7, MM4 and PADDD MM7, MM6 were moved from the bottom of the loop to the top (they were also copied below the loop). Two instructions have been added above the loop to initialize MM4 and MM6 for the first loop iteration.

Memory Operands

The PMADDWD instructions operate on data which is be loaded from memory. There are two possible instruction sequences to do this:

```
MOVQ reg1, mem1
MOVQ reg2, mem2
PMADDWD reg1, reg2
```

This sequence uses register operands only.

```
MOVQ reg1, mem1
```

```
PMADDWD reg1, mem2
```

In this sequence, the PMADDWD instruction uses a memory operand.

Unlike scalar IA instructions, MMX instructions with a memory operand suffer no penalty (if the operand is present in the L1 cache). Therefore the choice between the two options is not trivial and should be made on a case-by-case basis. Usually the second sequence is faster, since it contains one less instruction, but in some cases the first sequence can enable better pairing and lower clock counts. This is because the PMADDWD REG1, MEM2 instruction is restricted in pairing both by being a memory-access instruction and by being dependent on REG1. Splitting this into two instructions, each of which has only one of these restrictions, may facilitate pairing. In this code section, three out of four PMADDWD instructions use register operands; one uses a memory operand.

The loop overhead (four instructions) requires two clocks. It could be done with two less instructions (if we use ECX as an index register in addition to the EAX/EBX base register, the incrementation of EAX and EBX can be eliminated). However, since the first loop instruction would then have a one-clock AGI delay, no speed would be gained. If desired, the loop overhead can be reduced by further loop unrolling.

Data Alignment

Note that VDPMMX16 does not align the input vectors. The input vectors should both be aligned to eight bytes beforehand to avoid losing significant performance due to misaligned accesses (an additional three clocks per memory access). Note also that since the loop has been unrolled, VDPMMX16 can only operate on vectors if their size is a multiple of 16 words. For applications where this is not always the case, two possible solutions are:

- Pad the vectors with zeros until their size is a multiple of 16 before calling VDPMMX16.
- Alter VDPMMX16 to operate on vectors of all lengths. One way to do this is to add additional loops before and after the core loop. For an example of adding loops to handle vectors of arbitrary size and alignment, see application note AP-560, "Vector Arithmetic and Logic Operations".

3.0. PERFORMANCE GAINS

This section details the performance improvement as compared with traditional scalar code. There is approximately a 5X speedup for the MMX technology version. The results presented here assume all data is in the L1 cache and aligned to eight bytes - gains are reduced if there are cache misses or misaligned accesses.

3.1. Scalar Performance

Since the floating-point multiplication is significantly faster, the fastest scalar implementation is in floating-point. (Unless the vector is short enough for the overhead of converting between formats to be larger than the gain). Ignoring conversion overhead, a well-optimized implementation should be able to issue one multiplication, add or load per clock, so 16 elements + two clocks loop overhead should take about 50 clocks per iteration for a loop which operates on 16 elements.

3.2. MMX™ Code Performance

The inner loop executes in 10 clocks, which is five times faster than the scalar loop. For vectors which are much longer than 16 elements, the total performance improvement for the vector dot-product operation should be close to 500%, assuming all data is in the cache and that both vectors are aligned to eight bytes. If there is significant overhead for converting the data from integer to floating point, the speedup will be higher. The speedup is mostly attributable to the MMX instructions which perform operations on multiple data elements and can be paired, unlike the scalar instructions.

Peak MMX technology rate for multiply-accumulate operations is two 16-bit multiply-accumulate operations per clock. Peak scalar rate for multiply-accumulate operations is one per three clocks, so the peak speedup is 600%. This is reduced to 500% by the loop overhead. To achieve higher speedups the loop can be further unrolled. Though this reduces the loop overhead, care must be taken that the performance loss caused by code size increase is not larger than the gain.

4.0. VDPMMX16 FUNCTION: CODE LISTING

```
.486P
.model FLAT
PUBLIC    _vprod_mmx
_DATA SEGMENT
_DATA ENDS
_TEXT SEGMENT
_vprod_mmx PROC NEAR
src1      EQU       [esp+16]
src2      EQU       [esp+20]
vecsize   EQU       [esp+24]
result    EQU       [esp+28]
          push      ebx
          push      esi
          push      edi
          mov       eax, src1 ; src1 pointer (1 clock AGI delay)
          mov       ebx, src2 ; src2 pointer
          mov       ecx, vecsize       ; size of src1 and src2 arrays
          mov       edx, result        ; pointer to the result
          pxor      mm4, mm4  ; prepare for early use (SW pipelining)
          pxor      mm6, mm6  ; prepare for early use (SW pipelining)
          pxor      mm7, mm7  ; initialize accumulator
          ;.align 16
vdp16:
          movq      mm0, 0[eax]        ; first 4 source1 elements
          paddd     mm7, mm4  ; accumulate (from prev. iteration - SW pipelining)
          movq      mm1, 0[ebx]        ; first 4 source2 elements
          paddd     mm7, mm6  ; accumulate (from prev. iteration - SW pipelining)
          movq      mm2, 8[eax]        ; next 4 source1 elements
          pmaddwd   mm0,mm1            ; s1[0]*s2[0] + s1[1]*s2[1]::s1[2]*s2[2] + s1[3]*s2[3]
          movq      mm3, 8[ebx]        ; next 4 source2 elements
                                       ; V-pipe empty - memory can be accessed only in U-pipe
          movq      mm4, 16[eax]       ; next 4 source1 elements
          pmaddwd   mm2, mm3  ; s1[4]*s2[4] + s1[5]*s2[5]::s1[6]*s2[6] + s1[7]*s2[7]
          movq      mm5,16[ebx]        ; next 4 source2 elements
          paddd     mm7, mm0  ; accumulate
          movq      mm6,24[eax]        ; last 4 source1 elements
          pmaddwd   mm4, mm5  ; s1[8]*s2[8] + s1[9]*s2[9]::s1[10]*s2[10] + s1[11]*s2[11]
          pmaddwd   mm6, 24[ebx]       ; s1[12]*s2[12] + s1[13]*s2[13]::s1[14]*s2[14] + s1[15]*s2[15]
          paddd     mm7, mm2  ; accumulate
          add       eax, 32            ; increment source1 index
          add       ebx, 32            ; increment source2 index
          sub       ecx, 16            ; (NOTE: could elim. prev. pair, but AGI)
          jnz       vdp16
          paddd     mm7, mm4  ; accumulate
                                       ; V-pipe empty (mm6 locked because of pmaddwd)
          paddd     mm7, mm6  ; accumulate
                                       ; V-pipe empty (data dependency)

          movq      mm0, mm7  ; copy accumulator
                                       ; V-pipe empty (data dependency)
          psrlq     mm0,32             ; shift high order 32 bits of accumulation
                                       ; V-pipe empty (data dependency)
          paddd     mm7, mm0  ; add high and low order 32 bits of accumulation
                                       ; V-pipe empty (data dependency)
                                       ; One cycle stall - Op-Store dependency
          movd      [edx],mm7 ; store result
                                       ; V-pipe empty (integer inst. does not pair w/ MM memory reference)
          emms
          pop       edi
          pop       esi
          pop       ebx
          ret       0
_vprod_mmx ENDP
_TEXT     ENDS
END
```