Optimizing Code with MMX™ Technology

Mike Schmit mschmit@zoran.com mschmit@ix.netcom.com

Author

This article was written by Mike Schmit for the 1997 CGDC. Mike has been developing tools for optimizing x86 code in assembly language for over 10 years. At the time the article was written he was the Software Manager for CompCore Multimedia (later bought by Zoran Corp.). Mike worked with early MMX machines to develop fast MPEG-2 software decoders. Later this became part of the first software-only DVD player, SoftDVD.

Target Audience

This session is intended for developers considering using MMX Technology to optimize their applications for speed.

Introduction

MMX is a new technology that is an extension to the Intel x86 architecture used in the Pentium, 486 and previous processors. From a programming point of view, there are 57 new instructions, the MMX instructions. However, the internal operation of the pipeline, cache and a number of other features have also been changed. Future CPUs from Intel will also include these new MMX instructions. Intel has trademarked the term MMX Technology and it has been licensed by AMD for future CPUs. Other companies may also decide to license the MMX Technology.

Processors with the MMX instructions define a new set of eight 64-bit registers that are aliased onto the existing FPU registers. Prior to MMX, the FPU operated on the eight FPU registers as a LIFO (last-in, first-out) data stack with each register holding an 80-bit value. When these same registers are used as MMX integer registers they are treated as eight distinct 64-bit registers, randomly addressable as MM0 through MM7.

The 57 new instructions are simple, primitive instructions, although you may have heard that they were designed for multimedia and communication applications. Yes, they are very useful when processing video, audio and graphical data; but they are still simple single-cycle instructions (except the multiply instructions).

The real power of the new instructions is that they are SIMD (Single Instruction Multiple Data) instructions. What this means is that you can load, store or operate on multiple data items of the same type at the same time with a single instruction. That is the primary reason that these instructions are oriented to the highly parallel, repetitive sequences often found in multimedia operations.

The Pentium Processor with MMX Technology (Intel code name: P55C) is just the first Intel processor that will include MMX. The Pentium II (Intel code name: Klamath) is a chip with the PentiumPro (P6) micro-architecture and it includes the same MMX instructions. All future members of the Intel x86 family will also include MMX with the possible exception of low-cost versions for embedded systems, etc.

This paper discusses the MMX instructions in general and is an introduction to optimizing for performance on the P55C, the first shipping processor with the MMX instructions. Information on the Pentium II is also provided, having been just made public as this paper was written.

General Description

The MMX Pentium (my slang for the unwieldy official name: Pentium Processor with MMX Technology) is a superscalar (dual-pipeline) 32-bit CPU with 32 Kbytes of on-chip cache (16 Kbytes code, 16 Kbytes data). There are two six-stage integer pipelines named the U and the V pipelines. There are two nine-stage MMX pipelines and a single nine-stage floating-point pipeline. The first five stages of all of these are shared so that there are only two pipelines in total.

The main differences from the standard Pentium is the additional MMX sub-pipelines and the addition of a pipeline stage after the pre-fetch stage to decode instruction prefixes. This stage also contains a four-deep FIFO (first-in, first-out) buffer, decoupling the instruction fetch operation from the rest of the pipeline. This FIFO is new on the MMX Pentium.

Although the integer, MMX and floating-point execution units are capable of independent operation, a maximum of two integer instructions, two MMX instructions, one integer and one MMX instruction or a single floating-point instruction may be issued during a single clock cycle (there is one exception to this rule for floating point: FXCH). There are restrictions on the various combinations of integer and MMX instructions that may be issued together. For a complete description of the standard Pentium pairing, optimization guidelines and FPU operation see my book, Pentium Processor Optimization Tools, AP Professional, 1995.

Using MMX for DSP Applications

MMX adds a number of DSP-oriented features to the Intel architecture. For example, there is an instruction that performs four 16-bit integer multiplies and then adds adjacent 32-bit results. This can be used for a complex multiply or a vector product. Since the eight 64-bit MMX registers are aliased to the existing eight 80-bit floating point registers, this makes it somewhat inefficient to switch back and forth between the two modes. All other CISC-like attributes of the Pentium are still clearly visible, such as the addressing modes, small register set and irregular instruction set. For a more complete analysis of the DSP-like features of the Pentium with MMX Technology, contact Berkeley Design Technology at http://www.bdti.com or info@bdti.com.

Branch Prediction

Branch prediction on the MMX Pentium is implemented in the same manner as on the more advanced architecture of the Pentium Pro (P6) not like on the standard Pentium. The branch target buffer (BTB) stores the program counter (PC) address and target address of previously seen branches. When a branch instruction is fetched the BTB supplies the predicted target address. The prediction algorithm includes four history bits per target address that provides predictions based on the pattern of the previous four executions of the instruction. When no BTB entry has been made yet, such as the first execution of a conditional branch at the bottom of a loop, a static prediction model is used. The static model predicts forward branches as not
taken and backwards branches as taken.

Branch prediction improves the overall speed of programs compared to an equivalent CPU designed without it. However, branch prediction makes it somewhat difficult to accurately predict the number of clock cycles for any given execution and increases the range of possible execution times.

The MMX Pentium also contains a new feature called a return stack buffer (RSB). This allows accurate prediction of the return address from a function call, even though the call may be from varying addresses. This allows a loop with a function call to be unrolled without incurring a mispredicted branch on each return statement.

The MMX Memory System

The MMX Pentium has a single 32-bit address space (ignoring the System Management Mode) with two 16 Kbyte on-chip level one (L1) caches. Both caches are 4-way set associative with 32-byte line sizes. The data cache can be configured as write-through or write-back on a line-by-line basis. The MMX Pentium has four 64-bit write buffers, each available for either pipeline. The delay for a cache miss is eight CPU clock cycles. All other memory features are identical to the standard Pentium. The write buffers on the standard Pentium are dedicated to a particular pipeline.

Because of the write buffers between the CPU and the L1 cache and the write-back buffers between the L1 cache and L2 cache or main memory, there is a difference between sustained memory throughput and the throughput that can be attained for small data sizes. Applications must be tested with complete data sizes to insure the measured throughput is attainable.

The Pentium II memory system is quite different. The Pentium II L1 caches are the same size but are write-allocate. This means that when a memory write misses the cache a line fill operation is performed to fill the cache. In addition, the out-of-order execution capabilities of the Pentium II (P6-family) allow cache misses to not block execution within the CPU. The L2 cache is on a processor card and operates at 1/2 the CPU speed. At current clock speeds, this is a substantial improvement over the maximum 66mhz of the Pentium caches; but is slower than the full-speed L2 cache of the Pentium Pro (P6). The L2 cache can have 4 outstanding cache misses and the memory bus can have 8 outstanding requests. There is also a 12 deep load buffer once there are 4 outstanding cache requests.

Pipeline Operation

The Pentium uses a five-stage integer pipeline and an eight-stage floating-point pipeline. The MMX Pentium uses a six-stage integer pipeline and a nine-stage MMX/floating point pipeline. The integer stages are as follows:

PF	pre-fetch
F	fetch, decode prefix, queue in FIFO
D1	instruction decode
D2	address generation
EX	execute and cache access
WB	write back

The FPU pipeline has nine stages as follows:

PF	pre-fetch
F	fetch, decode prefix, queue in FIFO
D1	instruction decode
D2	address generation
EX	cache access; register read; FP data conversion for store
X1	FP Execute stage one; Conversion of FP load data
X2	FP Execute stage two
WF	Rounding and write FP results to register file
ER	Error reporting, status word update

The MMX pipeline has nine stages as follows:

PF	pre-fetch
F	fetch, decode prefix, queue in FIFO
D1	instruction decode
D2	address generation, read source
EX	committed for execution
Mex	Execute MMX ALU or shift/pack/unpack or 1st clock of multiply
Wm/M2	Write-back of single cycle ops, 2nd clock of multiply
M3	3rd clock of multiply
Wmul	Write-back of multiply result

Instructions are issued to the two pipelines in program order. As instructions are fetched, the CPU determines whether two instructions can be issued simultaneously or only a single instruction can be issued based upon the type of instructions and possible data conflicts. There are a number of rules which govern instruction pairing (i.e. issuing of two instructions in the same CPU cycle).

The MMX Pentium follows nearly the same pipeline pairing rules as the standard Pentium for integer and FPU instructions. MMX instructions have similar, but different rules. The rules for pairing on the Pentium are as follows:

Instruction pairing rules for Pentium

1. Both instructions must be simple. (See below)

2. Shifts or rotates can only pair in the U pipe.

3. ADC and SBB can only pair in the U pipe.

4. JMP, CALL and Jcc can only pair in the V pipe.

5. Neither instruction can contain BOTH a displacement and an immediate operand. For example:

```
mov     [ebx+2], 3     ; 2 is a displacement, 3 is immediate
mov     mem1, 4        ; mem1 is a displacement, 4 is immediate
```

6. Prefixed instructions can only pair in the U pipe. This includes extended instructions that start with 0Fh except for the special case of the 16-bit conditional jumps of the 386 and above. Examples of prefixed instructions:

```
mov    ES:[bx], 1
mov    eax, [si]       ; 32-bit operand in 16-bit code segment
mov    ax, [esi]       ; 16-bit operand in 32-bit code segment
```

7. The U pipe instruction must be only 1 byte in length or it will not pair until the second time it executes from the cache.

8. There can be no read-after-write or write-after-write register dependencies between the instructions except for special cases for the flags register and the stack pointer (rules 9 and 10).

```
mov    ebx, 2          ; writes to EBX
add    ecx, ebx        ; reads EBX and ECX, writes to ECX
                       ; EBX is read after being written, no pairing

mov    ebx, 1          ; writes to EBX
mov    ebx, 2          ; writes to EBX
                       ; write after write, no pairing
```

9. The flags register exception allows an ALU instruction to be paired with a Jcc even though the ALU instruction writes the flags and Jcc reads the flags. For example:

```
cmp    al, 0           ; CMP modifies the flags
```

```
    je      addr        ; JE reads the flags, but pairs

    dec     cx          ; DEC modifies the flags
    jnz     loop1       ; JNZ reads the flags, but pairs
```

10. The stack pointer exception allows two PUSHes or two POPs to be paired even though they both read and write to the SP (or ESP) register.

```
   push   eax          ; ESP is read and modified
   push   ebx          ; ESP is read and modified, but still pairs
```

Simple Instructions (for Pentium pairing)

Instruction format	16-bit example	32-bit example
MOV reg, reg	mov ax, bx	mov eax, edx
MOV reg, mem	mov ax, [bx]	mov eax, [edx]
MOV reg, imm	mov ax, 1	mov eax, 1
MOV mem, reg	mov [bx], ax	mov [edx], eax
MOV mem, imm	mov [bx], 1	mov [edx], 1
alu reg, reg	add ax, bx	cmp eax, edx
alu reg, mem	add ax, [bx]	cmp eax, [edx]
alu reg, imm	add ax, 1	cmp eax, 1
alu mem, reg	add [bx], ax	cmp [edx], eax
alu mem, imm	add [bx], 1	cmp [edx], 1

where alu = add, adc, and, or, xor, sub, sbb, cmp, test

INC reg	inc ax	inc eax
INC mem	inc var1	inc [eax]
DEC reg	dec bx	dec ebx
DEC mem	dec [bx]	dec var2
PUSH reg	push ax	push eax
POP reg	pop ax	pop eax
LEA reg, mem	lea ax, [si+2]	lea eax, [eax+4*esi+8]
JMP near	jmp label	jmp lable2
CALL near	call proc	call proc2
Jcc near	jz lbl	jnz lbl2
NOP	nop	nop
shift reg, 1	shl ax, 1	rcl eax, 1
shift mem, 1	shr [bx], 1	rcr [ebx], 1
shift reg, imm	sal ax, 2	rol esi, 2

shift mem, imm   sar ax, 15   ror [esi], 31

Notes:
- rcl and rcr are not pairable with immediate counts other than 1
- all memory-immediate (mem, imm) instructions are not pairable with a displacement in the memory operand
  - instructions with segment registers are not pairable

MMX pairing

All MMX instructions are considered "simple" instructions per the standard Pentium instruction pairing rules. MMX instructions can be paired with integer instructions or other MMX instructions. However, MMX and integer instructions have some pairing limitations, as described below:

- MMX instructions that access memory or integer registers must execute in the U pipeline.

- If there are two MMX instructions, they must use different MMX execution units. There are two MMX ALUs, but only one multiplier, one barrel shifter (used for shifts, packs and unpacks) and one memory access/integer access unit. The multiplier unit is fully pipelined so that a new multiply can be started on each CPU clock cycle, even though multiplies have a three cycle latency.

  - Because MMX instructions use a prefix byte (0fh) the decoder has been modified to allow pairing if the U pipe instruction is up to eleven bytes in length. (This changes rule 5 above).

There are a number of pipeline stalls that must be taken into account when writing optimum code for the Pentium or the MMX Pentium:

- Data-cache memory bank conflict. A one-cycle stall occurs if instructions in both pipelines attempt to access data in the L1 cache that are in the same memory bank (address line bits 2-4 are equal). (The V pipeline instruction stalls on the memory access while the U pipeline instruction continues. The U pipeline stalls in the previous stage, since no out of order execution is allowed.) This can only happen on integer instructions since the MMX instructions that access memory must only be in the U pipe.

- Address generation interlock (AGI). A one cycle delay occurs if an instruction in the previous cycle modified a register used in the address generation of an instruction in the current cycle.

- Prefix byte delay. Prefixed instructions on the Pentium cause a one cycle delay for each prefix. The MMX Pentium (sometimes) removes this delay by the addition of the F (fetch) stage in the pipeline and the FIFO buffer between the second and third pipeline stages. If the FIFO buffer is empty and the next instruction contains a prefix then there can be a one to three cycle stall while the next instruction(s) prefixes are decoded.

MMX Instruction set summary

Class	Instructions	Description
arithmetic	PADDx	packed add of bytes, words or dwords
	PADDSx	packed add with signed saturation of bytes or words
	PADDUSx	packed add with unsigned saturation of bytes or words
	PSUBx	packed sub of bytes, words or dwords
	PSUBSx	packed sub with signed saturation of bytes or words
	PSUBUSx	packed sub with unsigned saturation of bytes or words
	x = B (bytes), W (words), D (dwords)	
multiplication	PMULHW	multiply 4 pairs of words, keep high 16 bits of result
	PMULLW	multiply 4 pairs of words, keep low 16 bits of result
	PMADDWD	multiply 4 pairs of words, sum the first 2 and last 2 resulting in 2 32-bit results
comparison	PCMPEQx	compare for equality, set bits to 1 if equal, else 0
	PCMPGTx	signed compare for greater than, set bits to 1 if greater than, else 0
	x = B (bytes), W (words), D (dwords)	
logical	PAND	64-bit bitwise AND
	PANDN	64-bit bitwise AND then NOT
	POR	64-bit bitwise OR
	PXOR	64-bit bitwise XOR
pack/unpack	PACKUSWB	pack words to bytes with unsigned saturation
	PACKSSxx	pack with signed saturation words to bytes or dwords to words
	xx = WB (words to bytes), DW (dwords to words)	
	PUNPCKHyy	interleave high data from two operands

	PUNPCKLyy	interleave low data from two operands
	yy = BW (bytes to words), WD (words to dwords), DQ (dwords to qword)	
shift	PSLLz	shift left logical
	PSRLz	shift right logical
	PSRAz	shift right arithmetic (preserve sign)
	z = W (words), D (dwords), Q (qword)	
load/store	MOVQ	load/store 64-bits to/from memory or reg-reg
	MOVD	load/store 32-bits to/from memory or integer register
other	EMMS	empty MMX state (restore for FPU usage)

Notes:

All instructions in the arithmetic, multiplication, comparison, logical and pack/unpack groups (in the table above) use two operands. The first operand is an MMX register that is one source operand and the destination MMX register. The second operand is the second source operand and may be an MMX register or a memory reference. The shift instructions work the same except the second operand (the amount to shift by) may be an MMX register, memory reference or an immediate 8-bit constant.

The MOVQ and MOVD instructions work just like the traditional MOV. The EMMS instruction has no operands.

Detecting the Presence of MMX

Detecting the existence of MMX technology is done by executing the CPUID instruction and checking a set bit. This gives developers the flexibility to determine the specific code in their software to execute. During install or run time the software can query the processor to determine if MMX technology is supported and install or execute the code that includes, or does not include, MMX instructions as required. The following code tests for MMX:

```
;-------------------------------------------
; Check to see if the processor supports MMX
;
; returns: edx  0        no MMX
;               800000h  MMX available
;-------------------------------------------
```

```
      pushfd
      pop   eax               ; get eflags
      mov   edx, eax          ; copy of eflags
      push  eax
      popfd
      xor   eax, 00200000h    ; switch bit 21 (check for CPUID)
      pushfd
      pop   eax
      cmp   eax, edx          ; see if it can be changed
      jz    not_avail
      mov   eax, 1
      cpuid
      and   edx, 800000h      ; check MMX bit
      jz    not_avail
      ret

not_avail:
      xor   edx, edx          ; CPUID not supported or no MMX
      ret
```

Cache Operation

The SIMD nature of the MMX instructions improve the CPU performance by 2x to 8x on cached data operations. But that is the catch...the data must be cached to sustain this performance advantage. Algorithms and data structures must be carefully selected to keep the memory throughput high.

Caches are designed based on the principles of spatial and temporal locality. That is, they *expect* two things: once a data item is read, items in the same area will also be read; once a data item is read it is likely to be read again within a short amount of time.

All caches on the Pentium and above use a 32-byte cache line size. Each cache line consists of four quadwords (qword). When a cache miss occurs an entire cache line is brought into the cache from external memory (or the next level of cache). This is called a line fill. On the Pentium and above this data arrives in a burst composed of four quadwords. The burst operation timing is described as a sequence of four numbers, such as 3-1-1-1. This means that the first qword takes 3 cycles to read and that each of the next 3 qwords takes one additional cycle. The 4 qwords always start with the qword that contains the data the CPU is requesting. You might think that the next qword would be at the next higher sequential qword address -- but it isn't. It is important to know line fill sequence order, especially when processing data in a non-sequential order. The line fill order is as follows:

0, 1, 2, 3	(normal operation when progressing in forward order)
1, 0, 3, 2	(this one is counter-intuitive)

```
2, 3, 1, 2
3, 2, 1, 0        (backwards order)
```

So, if the first qword needed is located at qword 2 within a cache line, then (from the table above) we can see that the fill order is 2, 3, 1, 2.

The P6-family processors have a "write allocate by read-for-ownership" cache, whereas the Pentium and MMX Pentium have a "no-write-allocate; write through on write miss" cache.

So on a P6 when a write occurs and the write misses the cache, the entire 32-byte cache line is read. On the Pentium and MMX Pentium when the same write miss occurs, the write is simply sent out to memory. Write allocate is generally advantageous, since sequential stores are merged into burst writes, and the data remains in the cache for use by later loads. However, write allocate can be a disadvantage in code where:

- Only one element in a cache line is written
- Strides are larger than the 32-byte cache line
  - Only portions of a cache line are read after writing

The bottom line is that you must know your data and your algorithm and be sure to match it with the cache policy.

Instruction Examples

More examples and descriptions of each MMX instruction can be obtained on the Intel web sites (www.mmx.com and www.intel.com).

16-bit vector dot product

The following code section is the unoptimized inner loop for a dot product routine. The code takes 6 cycles to execute for every 4 input pairs (cached) or 1.5 cycles per element.

```
      xor         esi, esi                ; init index
      mov         ecx, count/4
      pxor        mm7, mm7                ; init sum to 0

loop1:                                    ; cycle
      movq        mm0, buffer1[esi]       ; 1

      pmaddwd     mm0, buffer2[esi]       ; 2, 3, 4

      paddd       mm7, mm0                ; 5
      add         esi, 8                  ; 5

      dec         ecx                     ; 6
      jnz         loop1                   ; 6 cycles per 4 elements

      movq        mm6, mm7
      psrlq       mm7, 32
      paddd       mm6, mm7                ; sum high and low results
      movq        mmword result, mm6
```

Unrolling the loop allows some overlap for the multiply instructions, as follows:

```
      xor         esi, esi                ; init index
      mov         ecx, count/8
      pxor        mm7, mm7                ; init sum to 0

loop1:                                    ; cycle
      movq        mm0, buffer1[esi]       ; 1

      pmaddwd     mm0, buffer2[esi]       ; 2, 3, 4

      movq        mm2, buffer1[esi+8]     ; 3

      pmaddwd     mm2, buffer2[esi+8]     ; 4, 5, 6

      paddd       mm7, mm0                ; 5

      paddd       mm7, mm2                ; 7
      add         esi, 16                 ; 7

      dec         ecx                     ; 8
      jnz         loop1                   ; 8 cycles per 8 elements

      movq        mm6, mm7
      psrlq       mm7, 32
      paddd       mm6, mm7                ; sum high and low results
      movq        mmword result, mm6
```

This allows the code to execute at 8 cycles per 8 data pairs: 1 cycle per pair (when cached). This might seem near optimum. However, you should always look at the code and determine the fastest possible speed, commonly referred to as the *speed of light*. In this case there are 3 operations for every 4 data pairs: a load, a load/multiply/add and an add. All other operations are loop overhead. (Note that we are taking advantage of the fact that PMADD can do a load and multiply with a single-cycle throughput.) With perfect instruction pairing this would be 0.375 (3/8) cycles per data pair.

The following code unrolls the loop again. But this time we do the loads for one iteration at the end of the loop.

```
      xor         esi, esi                ; init index
      mov         ecx, count/16

      pxor        mm7, mm7                ; init sum to 0

      movq        mm0, buffer1[esi]       ; pre-load from buffer1
      pxor        mm2, mm2                ; init to 0

      movq        mm1, buffer1[esi+8]     ; pre-load second items
      pxor        mm3, mm3                ; init to 0

loop2:                                    ; cycles
      pmaddwd     mm0, buffer2[esi]       ; 1
      paddd       mm7, mm2                ; 1

      pmaddwd     mm1, buffer2[esi+8]     ; 2
      paddd       mm7, mm3                ; 2

      movq        mm2, buffer1[esi+16]    ; 3

      movq        mm3, buffer1[esi+24]    ; 4
      paddd       mm7, mm0                ; 4

      pmaddwd     mm2, buffer2[esi+16]    ; 5
      paddd       mm7, mm1                ; 5

      pmaddwd     mm3, buffer2[esi+24]    ; 6

      movq        mm0, buffer1[esi+32]    ; 7

      movq        mm1, buffer1[esi+40]    ; 8

      add         esi, 32                 ; 9
      dec         ecx                     ; 9

      jnz         loop2                   ; 10 cycles per 16 elements

      movq        mm6, mm7                ; copy of results
      psrlq       mm7, 32
      paddd       mm6, mm7                ; sum high and low results
      movq        mmword result, mm6
```

This version gets much closer to the maximum speed attainable: 5/8 cycles per data pair. Merging the data pointer and loop counter into one variable would reduce the loop overhead by one cycle. No further optimizations seem possible because we would have an MMX load instruction in every cycle except for the one loop processing cycle. So we've hit a memory bandwidth wall.

My Optimization Strategy (P55C)

- Organize data structures and algorithms for SIMD
- Use simple instructions
- Unroll loops
  - Schedule instructions for optimal pairing

Pentium II Optimization *

Optimizing code for the Pentium II (P6, or Pentium Pro with MMX Technology) is quite similar to optimizing for the P55C. However, since the Pentium II includes out-of-order execution and multiple execution units that are not tied to a particular pipeline (like the P55C U and V pipelines) it is not as important to "schedule" each instruction and worry about the "pairing" rules. The Pentium II does have just two MMX execution units, so most performance increases will be due to the uncoupling of the load and store operations from the MMX and integer operations. In general, optimizing for the P55C will provide good performance on both processors, but optimizing for just the Pentium II will not provide for optimum P55C performance.

Celeron Optimization *

The Celeron Processor has the same CPU core as the Pentium II. The initial 266 and 300 Mhz parts were the Pentium II CPU with no L2 cache. Later, the "300A" and the 333 Mhz and above contain an integrated L2 cache. The L2 cache is 128K (1/4 the Pentium II's 512K external L2 cache) and operates at full CPU speed, whereas the 512K Pentium II L2 cache operates at ½ the CPU speed. In general, this is about an even tradeoff. Some algorithms will benefit while others will suffer.

Pentium III Optimization *

The Pentium III has the same basic P6 internal architecture, but adds a number of new instructions. The most important is the addition is what Intel calls the "Streaming SIMD Extensions" or SSE. There are several parts to this. The most important is the addition of 8 new 128-bit wide registers. Each register can contain four 32-bit floating point values that can be operated on in SIMD fashion (although the internal processing actually submits two groups of two floating point operations on subsequent cycles). Next there are some additions to the basic MMX integer operations. Finally, there are the "streaming" instructions. These allow speculative loads and non-blocking stores to be performed, recognizing that cache effects and the memory sub-system are the most important things to control in some algorithms. Full information about SSE can be obtained at the Intel web site.

Some available tools that support MMX

- Microsoft Visual C++ 4.x and above
- Numega Technologies Soft ICE version 3.0 and above
  - Intel Vtune 2.0 and above

References

- Intel Architecture MMX Technology, Developer's Manual, Order number: 243006-001 (001 is the edition number)

  - Intel Architecture MMX Technology, Programmer's Reference Manual, Order number: 243007-001 (001 is the edition number)

  - Intel web sites: http://www.intel.com, http://www.mmx.com

  - For DSP info: www.bdti.com

* these items added after the CGDC ’97 to make the document more up-to-date.