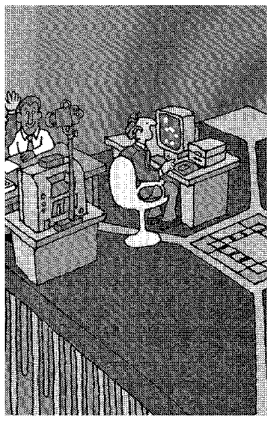
MMX Technology Extension to the Intel Architecture

Alex Peleg

Uri Weiser

Intel Israel Design Center



Designed to accelerate multimedia and communications software, MMX technology improves performance by introducing data types and instructions to the IA that exploit the parallelism in these applications.

MMX technology extends the Intel architecture (IA) to improve the performance of multimedia, communications, and other numeric-intensive applications. It uses a SIMD (single-instruction, multiple-data) technique to exploit the parallelism inherent in many algorithms, producing full application performance of 1.5 to 2 times faster than the same applications run on the same processor without MMX. The extension also maintains full compatibility with existing IA microprocessors, operating systems, and applications while providing new instructions and data types that applications can use to achieve a higher level of performance on the host CPU.[1]

We formed a group of microprocessor architects and select software developers that worked together to define MMX technology with clear guidelines and specific goals in mind. We aimed not only to improve performance and retain backward compatibility but also to design an innovative, state-of-the-art architecture that would scale with future higher frequency, advanced-microarchitecture processors.

We analyzed a wide range of software applications, including graphics, MPEG video, music synthesis, speech compression, image processing, games, speech recognition, and videoconferencing. The analysis results revealed that although the applications cross different domains, in most cases the computationally intensive, time-consuming routines within them have common characteristics. These are small native data types (for example, 8-bit pixels, 16-bit audio samples), localized recurring operations, and parallelism. This pointed us in the direction of a SIMD architecture that would exploit the available parallelism. The SIMD technique lets one instruction perform the same operation on multiple data elements, in parallel. It is the fundamental reason for the speedups achieved by MMX technology.

The imperative requirement that MMX technology retain full compatibility with existing operating systems and software placed some interesting constraints on our design. For example, we couldn't introduce new states, such as a register set, additional control registers, or new condition codes. Also, we couldn't permit new events (for example, new numeric exceptions) that existing operating systems would not acknowledge. Last but not least, we had to ensure that applications or operating systems would not rely on the invalid-opcode exception that present-day CPUs generate when executing the new MMX opcodes (for example, to internally signal some special event).

Basically, we used the existing floating-point registers as the MMX registers to maintain compatibility. We made this decision 1) to enjoy the existing 64-bit width possible with the floating-point registers and 2) as the main means to maintain full IA compatibility with existing applications and operating systems.

MMX technology and other processors

SIMD instructions have been implemented in other general-purpose processor architectures. The i860[2] and MC88110[3] processors implemented a limited set of SIMD graphics instructions targeted to support a few basic algorithms in 3D graphics, for example, Z buffering. More recently, the PA-7100LC processor[4] together with the SIMD instruction support in the PA-RISC 2.0 Architecture,[5] and the UltraSparc processors[6] have added a set of extensions to accelerate multimedia applications. Like MMX technology, both these architectures support parallel operations on multiple small packed data elements.

Like the UltraSparc design, MMX technology shares the floating-point registers between the floating-point and MMX instruc-

0272-1732/96/$5.00 © 1996 IEEE

tion sets.

To summarize, MMX technology employs

- data types of small data elements packed together into one register (we call them packed data types);
- an enhanced instruction set that operates on all data elements in a register in parallel, in a SIMD fashion;
- 64-bit MMX registers that are mapped on the IA floating-point registers; and
- full IA compatibility.

The MMX instructions are nonprivileged and usable in applications, libraries, and drivers. They require a minimal amount of incremental die area, making it practical to incorporate MMX technology into future Intel microprocessors.

Data types

MMX technology defines three packed (or compressed) data types and the 64-bit quadword. Each element within a packed data type is a fixed-point integer. Users control the place of the fixed point within each element and its placement throughout the calculation. While this adds a burden on users, it also gives them a large amount of flexibility. They can select and change fixed-point formats during the application course to fully control the dynamic range of their values.

MMX defines the following four data types (see Figure 1):

- packed byte, 8 bytes packed into one 64-bit quantity,
- packed word, 4 words packed into one 64-bit quantity,
- packed doubleword, 2 doublewords packed into one 64-bit quantity, and
- quadword, one 64-bit quantity.

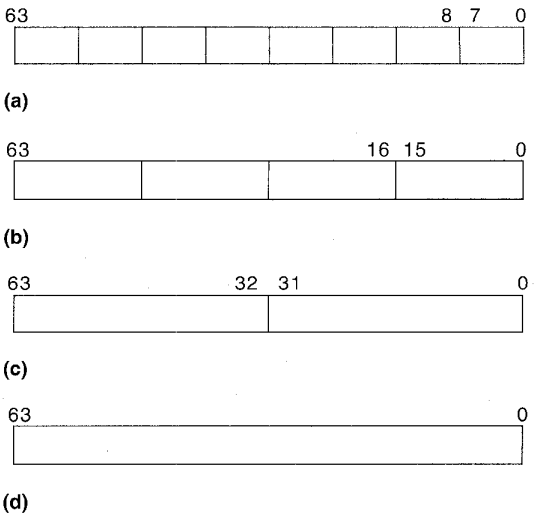


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

Enhanced instruction set

We defined a rich set of instructions that perform parallel operations on multiple data elements (8×8-, 4×16-, or 2×32-bit fixed point) packed into 64 bits. We defined full support for packed word (16-bit) data types. We had noted that the main data type in many multimedia algorithms is 16 bits, and it also serves as the higher precision backup for operations on byte data. To enable a wide variety of image algorithms, we supplied rich support for packed bytes. Basic support for packed doubleword data types helps operations that need higher precision than 16 bits, and a variety of 3D graphics algorithms. Overall, 57 MMX instructions were added to the IA.

Our instructions vary from one another by a few characteristics. The first is the data type on which they operate. Different instructions perform the same operation on different data types; for example, one instruction operates on a packed byte, and another operates on a packed word. Some instructions also vary in whether they treat values they operate upon as signed or unsigned.

A major feature of MMX instructions is saturation arithmetic. In regular fixed-point arithmetic when an operation overflows or underflows the register, we lose the most significant bits. For example, addition of two unsigned 16-bit numbers residing in a 16-bit register may result in an unsigned 17-bit result. This number is too large to be represented in a 16-bit register. The result's low-order 16 bits will appear in the result register; the seventeenth bit, however, will not fit and is usually truncated. This behavior generates a wraparound effect. In some architectures, a status flag (overflow and/or underflow indication) signals the occurrence of such a truncation.

With a saturating unsigned add instruction, instead of generating a 17-bit result and losing the seventeenth bit, the instruction result clamps to the largest possible unsigned number that can be represented in a 16-bit register, FFFFh. This operation is very important, for example, in algorithms dealing with visual data such as a 3D game implementing a Gouraud-shading technique.[7] This technique shades polygons by interpolating color values across scan lines. Along the way, calculations may start to overflow. Without precautions, this may generate, as a result of the wraparound effect, a completely different value than expected. A dark polygon being shaded toward the color black may suddenly acquire white pixels.

Saturation makes sure these kinds of problems do not occur. It also eliminates the need to check for overflows (or underflows), a time-consuming operation that can substantially slow down fast inner loops. In MMX technology, saturation is not a mode. It is not activated by setting a control bit or something similar. Simply, some instructions have saturation as part of their operation. For example, MMX add instructions vary; some use wraparound arithmetic, and others use saturating arithmetic.

We designed the MMX instructions to be fast and scale with higher frequencies and advanced microarchitectures. On the first implementation, a Pentium processor, all MMX instructions with the exception of the multiply instructions execute in one cycle. The multiply instructions have a three-cycle execution latency, but the multiply unit's pipelined design enables a new multiply instruction to start every cycle.

Table 1. MMX instruction set summary.

Opcode mnemonic*	Options	Cycle count	Description
PADD(b,w,d), PSUB(b,w,d)	Wraparound and saturate	1	Adds or subtracts packed 8 bytes, four 16-bit words, or two 32-bit doublewords in parallel.
PCMPEQ(b,w,d), PCMPGT(b,w,d)	Equal or greater than	1	Compares packed 8 bytes, four 16-bit words, or two 32-bit elements in parallel. Result is mask of 1s if true or 0s if false.
PMULLW, PMULHW	Result: high- or low-order bits	Latency: 3; throughput: 1	Multiplies four packed, signed 16-bit words in parallel. Chooses low- or high-order 16 bits of the 32-bit result.
PMADDWD	Word to doubleword conversion	Latency: 3; throughput: 1	Multiplies four packed, signed 16-bit words and adds together adjacent pairs of 32-bit results in parallel. Result is a doubleword.
PSRA(w,d), PSLL(w,d,q), PSRL(w,d,q)	Shift count in register or immediate	1	Shifts arithmetic right, logical left and right packed 4 words, 2 doublewords, or the full 64 bits (quadword) in parallel.
PUNPCKL(bw,wd,dq) PUNPCKH(bw,wd,dq)	—	1	Merges packed 8 bytes, four 16-bit words, or two 32-bit doublewords with interleaving.
PACKSS(wb,dw)	Always saturate	1	Packs doublewords to words or to bytes in parallel.
PAND, PANDN, POR, PXOR	—	1	Performs 64-bit bitwise logical operations.
MOV(d,q)	—	1 (if data in cache)	Moves 32 or 64 bits between memory and MMX registers; 32 bits can be moved between MMX and integer registers.
EMMS	—	Varies by implementation	Empty floating-point registers tag bits.

*If an instruction supports multiple data types, brackets indicate the data type: b = byte, w = word, d = doubleword, q = quadword; bw = from byte to word, wd = from word to doubleword, dq = from doubleword to quadword, wb = word to byte, dw = from doubleword to word.

This means that with software loop unrolling[8] the processor can achieve a throughput of one cycle per SIMD multiply.

Table 1 summarizes the types of instructions included in MMX and the operations they perform.

Packed addition and subtraction with optional saturation. These instructions exist for the three packed data types. Each single add or subtract operation is independent of the others; takes place in parallel; and comes in wraparound, unsigned-saturation, and signed-saturation versions. The upper and lower saturation limits are FFh and 00h for unsigned bytes and 7Fh and 80h for signed bytes. For words and doublewords, the limits are also the maximum and minimum unsigned or signed values that these data types can represent.

Packed multiplications. We defined two variations of these instructions, both of which support 16-bit-precision multiplications.

The first performs four 16-bit×16-bit multiplies and lets the user choose the low- or high-order parts of the 32-bit multiply result. Thus, the input operands and the results are packed 16-bit data types.

The second is the basis for the fast multiply-accumulate capability in MMX (see Figure 2). The idea was to design a 16-bit multiplier but perform accumulation in 32 bits to enable full accumulation precision. This instruction starts from a packed 16-bit data type and returns a packed 32-bit data type. It multiplies respective elements from both its sources, generating four 32-bit results. It adds two adjacent products for one 32-bit result, then adds the two other adjacent products to generate the final packed doubleword. Thus, the packed multiply-add instruction performs four multiplies and two 32-bit accumulations in one instruction. To complete the multiply-accumulate operation, a PADDD instruction adds the results to another register, which is used as the accumulator.

Packed compares. These instructions independently compare all the respective data elements of two packed data types, in parallel. They generate a mask of 1s and 0s, depending on whether the condition is true or false. There are no new condition code flags for the packed compares, and they do not affect existing IA condition code flags. Figure 3 shows an example of a compare greater-than operation on packed-word data.

Subsequent code can use the result mask of 1s and 0s to select elements from different inputs, when combined with logical operations. We explain the use of MMX compare instructions more fully in the later chroma key example.

Packed shift instructions. MMX implements two versions of logical left, right, and arithmetic right shift operations. The first is the regular packed shift that independently shifts each element in the packed data type in parallel. MMX supports shift operations on packed word and doubleword data types. Packed shift instructions let users control precision by having full control over the binary-point position in fixed-point values. We did not include packed byte shifts because our studies indicated that bytes are usually used at full precision and do not require frequent precision control.

The second version of shift operations is logical shift left or right on the whole 64-bit MMX register. These shift operations are especially important; they enable realignment of the packed data that was loaded from memory. MMX is based on moving 64-bit quantities around, and loading or storing them to memory. Because most IA CPUs have a sub-

stantial performance degradation on misaligned memory accesses, MMX code should perform only 64-bit aligned accesses. Also, most compilers and code writers do not enforce 64-bit alignment. For example, an MMX code developer might want to access an array of bytes, loading 8 bytes at a time but at an alignment that does not start on an address that divides by eight. To avoid performance degradation, the code developer can perform two adjacent aligned memory accesses of 64 bits each, and then, using the 64-bit MMX shift instructions, realign the data as needed.

Conversion instructions. The MMX pack and unpack instructions facilitate conversions between the packed data types. This is especially important when an algorithm needs higher precision in its intermediate calculations. For example, in an image-filtering operation the data is loaded as packed bytes—one byte per pixel in one of the color planes. The filter operation first requires multiplying the filter coefficient with a set of adjacent image pixels, and then adding the results—an operation that almost certainly will overflow 8 bits. To avoid the overflow problem, users can unpack the packed bytes into packed words, perform the whole operation in 16-bit precision, and then with one pack instruction go back to packed byte data and store to memory.

The unpack instruction unpacks from a smaller precision data type to a higher precision data type. However this instruction also serves another purpose: It performs an interleaved merge operation. Figure 4 is an example of an unpack instruction on packed byte data. Here, the instruction takes four low-order bytes from each input operand and interleaves them into the result register. This operation is useful in many instances such as

- interpolation operations in which a new pixel is required between every pair of old pixels,
- matrix transpositions (converting columns to rows), and
- conversions between RGB or RGBA-pixel format to and from color planes.

The special case of an input register that holds B*n* as all 0s achieves the effect of unpacking unsigned bytes of A*n* to unsigned words.

Logical operations. MMX technology adds a set of 64-bit logical operations: AND, ANDNOT, OR, and XOR.

Memory transfer. Since MMX technology deals with 64-bit quantities, we had to add new instructions to transfer packed data to and from memory. The move quadword (MOVQ) instruction moves 64-bit data between MMX registers and memory or between MMX registers and themselves. We also added a 32-bit memory move for packed data (MOVD) to transfer 32 bits of data between memory and MMX registers. This instruction always moves the low-order 32 bits of an MMX register. The register-to-register version of this instruction implements the operation of moving data between the MMX and integer register files.

Empty MMX state operation. We also added one instruction to aid users in maintaining compatibility with IA and existing software (see later section).

64-bit MMX registers

We provided eight 64-bit general-purpose registers that are actually the floating-point registers. Assembly code can directly address each register by designating MM0-MM7 register names in the MMX instructions. The registers are random-access registers; that is, they are not accessed via a stack model as with the floating-point instructions. The MMX registers hold only MMX data. MMX instructions that specify a memory operand use the IA integer registers to address that operand.

Thus, the following example is a legal MMX instruction:

Paddw mm4, 16[ebx]: Add packed word in memory to mm4

Note that one of the issues programmers have with the IA is that it contains a small number of general-purpose integer registers—even the eight IA integer registers are not all general purpose. (ESP, the stack pointer register, maintains a pointer to the top of the stack and is implicitly manipulated in many IA instructions that access the software stack. As a result, it cannot be used as a general-purpose register.)

Since the MMX registers are actually the floating-point registers, applications that use MMX instructions can use almost 16 registers. Eight are the MMX/floating-point registers, each of which are 64 bits in size and hold packed data. The other eight are integer registers available for different operations such as addressing, loop control, or other data manipulation. We found that this release of register pressure makes register allocation much simpler and saves many instances

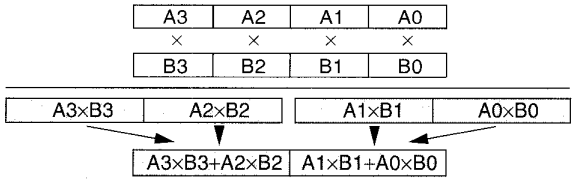


Figure 2. Packed multiply-add word to doubleword.

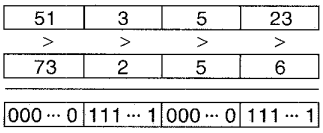


Figure 3. Packed compare greater-than word.

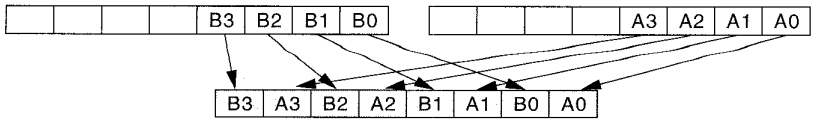


Figure 4. Unpacking byte data.

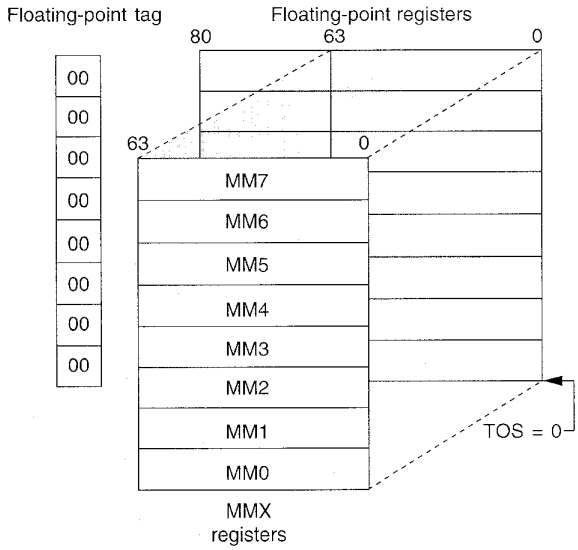


Figure 5. Mapping MMX registers to floating-point registers. TOS: top-of-stack field.

of having to store values temporarily to memory.

Full IA compatibility

MMX achieves full compatibility with existing operating systems and applications by using the existing IA floating-point state as temporary storage for MMX data (see Figure 5). We did not add registers, condition codes, or events to support MMX. With no mode switch or MMX mode indication, an existing operating system has nothing new to address.

Sharing the floating-point registers revealed several challenging definition issues. One is the elimination of the stack architecture of the floating-point register file for MMX register access. Access is random.

Another issue regards an application that requires use of MMX and floating-point codes. The dual usage of the floating-point registers does not preclude applications from using both MMX and floating-point code. Applications, though, should not attempt to use the registers simultaneously for floating-point and MMX data. Since floating-point register values are interpreted differently when accessed by floating-point and MMX instructions, users should not rely on register content across transitions between MMX and floating-point codes. Partitioning the floating-point and MMX codes into long execution spells makes transition events infrequent and simple to handle.

For all practical purposes, an existing operating system views MMX as an extension of floating-point instructions, not a new extension.

Mapping. MMX instructions write values to the low-order 64 bits (the mantissa) of the IA 80-bit floating-point registers. When an MMX instruction writes to a register, the exponent field of the corresponding floating-point register (bits 64 through 78) and the sign bit (bit 79) are set to 1s. This makes the value in the register a NAN (not a number) or infinity when viewed as a floating-point value. Since MMX instructions operate only on the low-order 64 bits of the floating-point registers, they are not affected by their generation of invalid floating-point values.

The software convention followed today by PC code writers is to leave the floating-point stack empty after using it. IA defines a tag field for each floating-point register. These fields signal when their corresponding register is empty, valid, or a special case of floating-point value (a NAN, for example). When an application tries to load a value into the floating-point stack beyond its capacity, the tag bits indicate the space problem and a stack overflow indication or exception is generated.

Following the convention ensures that subsequent floating-point code can start operating on an empty stack. When a new value is pushed onto the stack into a new floating-point register, the corresponding tag field becomes valid. When a value is popped off the stack and out of a floating-point register, the corresponding floating-point tag field changes to empty. This implicit empty/valid effect is a natural side effect of organizing the floating-point registers into a stack architecture.

An operating system can use these tags to perform "intelligent" state saving on context switches. For example, an operating system may compare the whole tag register (a register holding all tag fields) to all 1s (the empty indication). If it finds all registers empty, it doesn't save them.

Randomly accessed MMX registers make it difficult to implement the implicit effect that the floating-point instructions have over the floating-point tags. As a result, we decided to design the effect on the tags more explicitly. The first time an MMX instruction accesses a floating-point register, the tag bits of all the floating-point registers are validated. This ensures that all the registers are saved on a context switch. MMX also supplies the EMMS (empty MMX state) instruction to convert all the tag bits at once to the empty state. Programmers should insert EMMS after each MMX code section that may be followed by floating-point code. EMMS is the MMX means to conform to the software convention of leaving the floating-point stack empty before someone else's floating-point code may try to use it. EMMS affects only the tags, not the register content. So if the need arises, MMX instructions can continue to operate on the values in the registers after EMMS executes.

Table 2 summarizes the effects of the MMX instructions on the floating-point state.

Context switch support. To complete our compatibility needs, we made sure that MMX technology uses the same techniques used by the floating-point architecture to interface with the operating system.

For example, IA includes the concept of lazy task switching. If floating point is not used extensively, there is no point in saving and restoring the floating-point state on every task switch. The floating-point state must be saved only if more than one task uses floating point. IA implements this by supplying a bit that the operating system can set. This bit instructs the CPU to signal when any task tries to use floating point. By setting this bit, the first floating-point instruction that executes in a task will trigger an exception. This exception enables an operating system to save the context of a previous floating-point task only if another task wants to use floating point. Since MMX code uses the floating-point registers,

the same behavior extends to MMX instructions. The first MMX instruction executed after setting the bit for lazy task switching will also trigger the exception. This enables the operating system to save the floating-point state before the MMX code overwrites it.

The instructions an operating system uses to save and restore the floating-point state (FSAVE and FRSTOR) also save and restore the MMX values. These instructions save the values in the floating-point registers regardless of whether they are used by floating-point or MMX codes.

Enhancing existing applications. Executing the CPUID instruction and checking a set bit in the result detects the existence of MMX technology on Intel microprocessors. This gives software developers the flexibility of determining what software should be run. During runtime, the software can query the microprocessor to determine MMX support and execute regular IA or MMX code based on the answer. This enables the software developer to distribute one version of the product that will run on a CPU without MMX support, but run even better on a CPU with MMX. Binary-code size will need to grow to support this duality. Our studies show that since MMX code is mainly applied to small, tight, computationally intensive loops, and only two versions of these loops must be supplied, the overall binary-code growth is about 10 percent.

Table 2. Effects of MMX instructions on the floating-point state.

Instruction type	Floating-point tag word	Other FPU state	Exponent/ signed bits	Mantissa
MMX register read	All fields set to 00 (valid)	Unchanged	Unchanged	Unchanged
MMX register write	All fields set to 00 (valid)	Unchanged	Set to 1s	Overwritten
EMMS	All fields set to 11 (empty)	Unchanged	Unchanged	Unchanged

```
vector_x_matrix_4x4(MMX64 *v, MMX64 *m) {
MMX64 v0101,v2323,t0,t1,t2,t3;
   v0101 = punpckldq(v,v);      /* unpack v0 and v1 with themselves */
   v2323 = punpckhdq(v,v);      /* unpack v2 and v3 with themselves */
   t0 = pmaddwd(v0101,m[0]);    /* multiply v0 and v1 with first 2 rows */
   t1 = pmaddwd(v2323,m[1]);    /* multiply v2 and v3 with first 2 rows */
   t2 = pmaddwd(v0101,m[2]);    /* multiply v0 and v1 with last 2 rows */
   t3 = pmaddwd(v2323,m[3]);    /* multiply v2 and v3 with last 2 rows */
   t0 = paddd(t0,t1);           /* add first half of first rows with second half */
   t2 = paddd(t2,t3);           /* add first half of last rows with second half */
   v = packssdw(t0,t2);         /* pack the results from 32-bit to 16-bit with
                                   saturate */
}
```

Figure 6. Sample using MMX instruction intrinsics.

Accessing instructions from C

MMX technology is an assembly extension and requires coding in assembly. Though most software developers do not find it easy to code in assembly, they don't have to recode the whole application to enjoy the full benefits of MMX technology. Only the computationally intensive loops need to be recoded in MMX instructions. For example, in MPEG-2 video decoding,[9] substantial speedup comes from rewriting the IDCT (inverse discrete cosine transform)[10] with MMX instructions. IDCT has a well-defined interface and a relatively small code footprint. It also turns out that these small computationally intensive loops are the best place to employ MMX technology. The rest of the code in many cases does not lend itself to easy conversion to MMX and its parallel characteristics.

We also developed the concept of MMX instruction intrinsics. Intrinsics generate optimized MMX assembly, even while coding directly in C. Programmers still need to design the algorithm with MMX data types and dataflow in mind, but they can use a C level macro interface to abstract the MMX instructions. The main advantage of this technique is that programmers do not need to worry about register allocation or assembly code scheduling; the compiler takes care of these optimizations.

The example using intrinsics in Figure 6 shows how to multiply a 4×4 matrix by a four-element vector, resulting in a four-element vector. We assumed the input and matrix elements are 16-bit signed values. The result is a packed word that holds the result vector. To facilitate the exploitation of the available parallelism in this operation, we reordered the matrix elements as follows:

$$m[0] = (M_{00}, M_{01}, M_{10}, M_{11}),\ m[1] = (M_{02}, M_{03}, M_{12}, M_{13})$$
$$m[2] = (M_{20}, M_{21}, M_{30}, M_{31}),\ m[3] = (M_{22}, M_{23}, M_{32}, M_{33})$$

MMX64 type that appears in the code is compiler specific and can, for example, be an abstraction of the int64 data type or a structure with two long integers.

Duplicating a pair of vector elements to generate (V0, V1, V0, V1) and reordering the matrix elements enable each PMADDWD instruction to multiply simultaneously two rows of the matrix against the vector. Figure 7 shows the flow of the algorithm for the first two results. The same operations perform the next two results.

Performance examples

We can analyze the performance enhancement of MMX technology with an example of a matrix-vector multiplication very

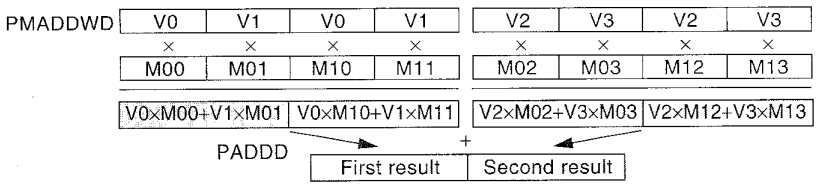


Figure 7. Flow diagram of matrix-vector multiply.

much like the one in Figure 6. This operation and similar ones appear in many multimedia algorithms and applications.

A multiply-accumulate operation (MAC)—the product of two operands added to a third operand (the accumulator)—requires two loads (operands of the multiplication operation), a multiply, and an add (to the accumulator). MMX does not support three-operand instructions, therefore it does not have a full MAC capability. On the other hand, MMX does define the PMADDWD instruction that performs four multiplies and two 32-bit adds. A following PADDD instruction performs the additional two adds.

We start by looking at a vector dot product, the building block of the matrix-vector multiplication. For this performance example, we assume both input vectors are 16 elements long, with each element in the vectors being signed 16 bits. Accumulation takes place in 32-bit precision. A Pentium processor microarchitecture, for example, would have to process the operations one at a time in sequential fashion. This amounts to 32 loads, 16 multiplies, and 15 additions, for a total of 63 instructions. Assume we perform four MACs (out of the 16) per loop iteration of our code. Then, we need to add 12 instructions for loop control (3 instructions per iteration, increment, compare, branch) and 1 instruction to store the result. Now the total is 76 instructions.

Assuming all data and instructions are in the on-chip caches, and that exiting the loop will incur one branch misprediction, the integer assembly optimized version of this code (using both pipelines) takes just over 200 cycles on a Pentium processor microarchitecture. The cycle count is dominated by the nonpipelined, 11-cycle integer multiply operation. Under the same conditions, but assuming the data is in floating-point format, the floating-point optimized assembly version executes in 74 cycles. This version is faster as the floating-point multiply takes only three cycles to execute and executes in a pipelined unit.

Now, we can look at MMX technology. MMX computes four elements at a time. This reduces the instruction count to eight loads, four PMADDWD instructions, three PADDD instructions, one store instruction, and three additional instructions (overhead due to packed data types), totaling 19 instructions. Performing loop unrolling of four PMADDWD instructions eliminates the need to insert loop control instructions. The four PMADDWDs already perform the 16 required MACs. Thus, the MMX instruction count is four times less than that for integer or floating-point operations. With the same assumptions applied to a Pentium processor microarchitecture, an MMX-optimized assembly version of the code using both pipelines will execute in only 12 cycles. This is a speedup of six times over floating-point and much more over integer.

Now, we extend this example to a full matrix-vector multiply. We assume a 16×16 matrix multiplies a 16-element vector, an operation built of 16 vector dot products. Repeating the same exercise as before, and assuming a loop unrolling that performs four vector dot products each iteration, the regular Pentium processor floating-point code will total 4(4×76 + 3) or 1,228 instructions. Using MMX technology will require 4(4×19 + 3) or 316 instructions. The MMX instruction count is 3.9 times less than when using regular operations. The best regular code implementation (floating-point optimized version) takes just under 1,200 cycles to complete in comparison to 207 cycles for the MMX code version. This is a speedup of 5.8 times.

Chroma keying

Chroma keying is an image overlay technique frequently referred to as the weatherman example. In this example, we use a dark-blue screen to overlay an image of a woman on a picture of a spring blossom (see Figure 8). The required C code operation is

```
    for (i=0; i<image_size; i++) {
if (x[i] == Blue)  new_image[i] =y[i];
        else new_image[i] = x[i];
    }
```

where x is the image of the woman on a blue blackground, and y is the image of the spring blossom.

Using MMX technology, we load eight pixels from the picture with the woman on a blue background. In Figure 9, the compare instruction builds a mask for that data. This mask is a sequence of byte elements that are all 1s or all 0s, representing the Boolean values of true and false. This reflects the "unwanted" background and what we want to keep. Figure 9 shows this result using a black-and-white picture.

Figure 10 shows this mask being used on the same eight pixels from the picture with the woman and the corresponding eight pixels from the spring blossom. The PANDN and PAND instructions use the mask to identify which pixels to keep from the spring blossom and the woman. They also turn the unwanted pixels to 0s. The POR instruction builds the final picture.

The MMX code sequence in Figure 11 processes eight pixels using only six MMX instructions and doing so without branches. Being able to process a conditional move without using branch instructions or looking up condition codes is becoming an important performance issue with the advanced, deep-pipeline microarchitectures that use branch prediction. A branch based on the result of a compare operation on the incoming data is usually difficult to predict, as incoming data in many cases can change randomly and thus degrade the prediction quality. Eliminating branches used for data selection, together with the parallelism of the MMX instructions, combines into an important performance enhancement feature.

Figure 8. Chroma keying: image overlay using a background color.

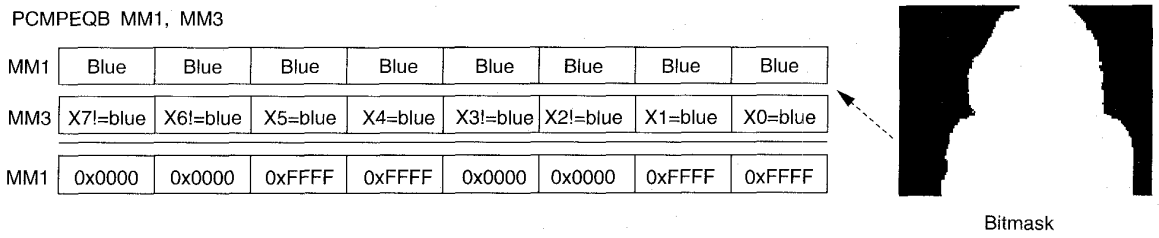


Figure 9. Generating the selection bit mask.

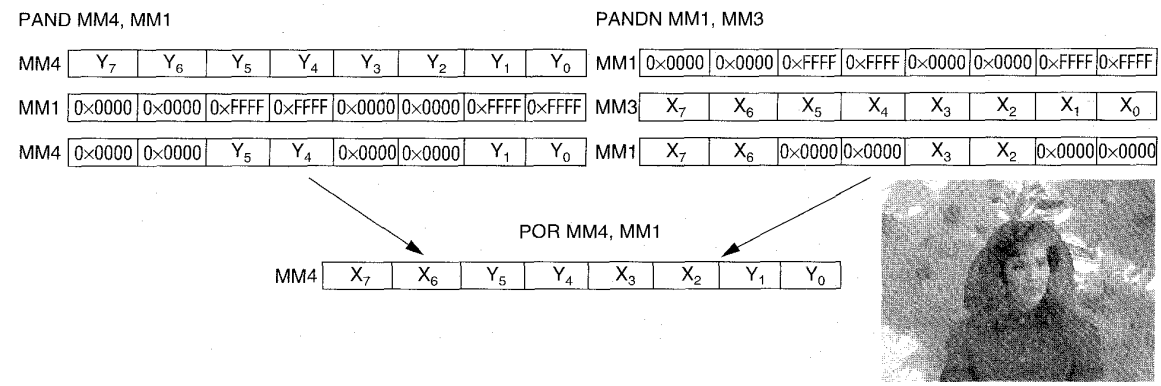


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

When unrolling the loop and scheduling it for a Pentium processor microarchitecture, the MMX code processes eight pixels in three cycles (assuming data is in the on-chip cache), that is, 3/8 cycles per pixel. Doing the same with the regular IA integer instruction set would require almost three cycles for a single pixel. (This assumes a high, 85 percent correct branch prediction.)

In general, the performance boosts of MMX code on Intel processors with MMX technology result from the combination of the following two features:

- the advantage in instruction count resulting from the multiple parallel operations performed in each SIMD

```
Movq    mm3, mem1   /* Load eight pixels from
                       woman's image
Movq    mm4, mem2   /* Load eight pixels from the
                       blossom image
Pcmpeqb mm1, mm3
Pand    mm4, mm1
Pandn   mm1, mm3
Por     mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.

MMX instruction, and
- exploiting parallelism between instructions via the advanced microarchitectural implementations of Intel processors (for example, superscalar execution).

MMX technology enhances applications that benefit from SIMD architecture and parallelism. MMX speeds up computationally intensive inner loops or subroutines on average between three to five times. When these are applied to the full application, that application typically runs on the same processor 1.5 to 2 times faster than the same application without MMX technology.

For example, a certain MPEG-1 video decoding application on a Pentium class processor with MMX technology executes 1.5 times faster than the same application on the same processor not using MMX technology. An assortment of image filters in an image-processing application execute just over four times faster.

INTEL PLANS TO IMPLEMENT MMX technology on future Pentium and Intel architecture processors. It will make MMX technology a base capability on all company CPUs to allow existing and new applications to run faster. We believe the performance gains from this technology will scale well with the CPU operating frequency and future Intel microarchitecture generations.

Acknowledgments

Many people helped develop MMX technology, especially Benny Maytal, David Bistry, Robert Dreyer, Carole Dulong, Steve Fischer, Andy Glew, Koby Gottlieb, Eiichi Kowashi, and Larry Mennemeier. We also thank a large team of architects and software developers at Intel for their help and support of the definition process and application analysis, especially Benny Eitan, Mike Keith, Oded Lempel, and Dave Sprague and his team. Special thanks to Adina Hagege who made this document readable (we hope).

References

1. M. O'Conner, "Extending Instructions for Multimedia," *Electronic Engineering Times*, No. 874, Nov. 1995, p. 82.
2. *i860TM Microprocessor Family Programmers Reference Manual*, order no. 240875-001, Intel Corporate Literature Sales, Mt. Prospect, Ill., 1991.
3. K. Diefendorff and M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," *IEEE Micro*, Vol. 12, No. 2, Apr. 1992, pp. 40-63.
4. R.B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, Vol. 15, No. 2, Apr. 1995, pp. 22-32.
5. R. Lee and J. Huck, "64-Bit and Multimedia Extensions in the PA-RISC 2.0 Architecture," *Proc. Compcon 96*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 152-160.
6. M. Tremblay et al. "The Visual Instruction Set (VIS) in UltraSPARC," *Proc. Compcon*, IEEE CS Press, 1995, pp. 462-469.
7. A. Watt, "3D Computer Graphics," 2nd ed., Ch. 5.3, Addison-Wesley, Reading, Mass., 1993, pp. 131-136.
8. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Mateo, Calif., 1990, pp. 315-318.
9. *ISO/IEC 13818, Generic Coding of Moving Pictures and Associated Audio; Part 1: System, Part 2: Video, Part 3: Audio*, Int'l Organization for Standardization/Int'l Electrotechnical Commission, Geneva, 1995.
10. C. Loeffler, A. Lightenberg, and G. Moshytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications," *Proc. 1989 Int'l Conf. Acoustics, Speech, and Signal Processing*, IEEE CS Press, 1989.

Alex Peleg is a senior computer architect in the Israel Design Center Architecture Group for Intel's operations at Haifa. He leads a team of architects dealing with the definition of multimedia architectures for CPUs. His interests include advanced CPU and multimedia microarchitectures.

Peleg received his BSc degree in computer science and MSc degree in electrical engineering from the Technion, Israel Institute of Technology. He is a member of the IEEE and the Association for Computing Machinery.

Uri Weiser manages the Computer Architecture Group for the Intel design operation in Haifa, Israel. He holds responsibility for the Pentium architecture, definition of multimedia architecture for CPUs, and x86 microarchitecture future research. He also holds an adjunct senior lecturer position at the Technion, Israel Institute of Technology. He has worked at the Israeli Department of Defense and later headed the design of the NS32532 processor at the National Semiconductor Design Center in Herzelia, Israel.

Weiser received his BSc and MSc degrees in electrical engineering from the Technion, Israel Institute of Technology, and his PhD in computer science from the University of Utah. He is an associate editor of *IEEE Micro* and a member of the IEEE and the Association for Computing Machinery.

Direct questions concerning this article to Alex Peleg, Intel Israel, PO Box 1659, Matam, Haifa 31015, Israel; apeleg@iil.intel.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 159 Medium 160 High 161