MMX™ Technology Architecture Overview

Millind Mittal, MAP Group, Santa Clara, Intel Corp.
Alex Peleg, IDC Architecture Group, Israel, Intel Corp.
Uri Weiser, IDC Architecture Group, Israel, Intel Corp.

Index words: MMX™ technology, SIMD, IA compatibility, parallelism, media applications

Abstract

Media (video, audio, graphics, communication) applications present a unique opportunity for performance boost via use of Single Instruction Multiple Data (SIMD) techniques. While several of the compute-intensive parts of media applications benefit from SIMD techniques, a significant portion of the code still is best suited for general purpose instruction set architectures. MMX™ technology extends the Intel Architecture (IA), the industry's leading general purpose processor architecture, to provide the benefits of SIMD for media applications.

MMX technology adopts the SIMD approach in a way that makes it coexist synergistically and compatibly with the IA. This makes the technology suitable for providing a boost for a large number of media applications on the leading computer platform.

This paper provides insight into the process followed for the definition of MMX technology and the considerations used in deciding specifics of MMX technology. It discusses features that enable MMX technology to be fully compatible with the existing large application and system software base for IA processors. The paper also presents examples that highlight performance benefits of the technology.

Introduction

Intel's MMX™ technology [1, 2] is an extension to the basic Intel Architecture (IA) designed to improve performance of multimedia and communication algorithms. The technology includes new instructions and data types, which achieve new levels of performance for these algorithms on host processors.

MMX technology exploits the parallelism inherent in many of these algorithms. Many of these algorithms exhibit the property of "fixed" computation on a large data set.

The definition of MMX technology evolved from earlier work in the i860™ architecture [3]. The i860 architecture was the industry's first general purpose processor to provide support for graphics rendering. The i860 processor provided instructions that operated on multiple adjacent data operands in parallel, for example, four adjacent pixels of an image.

After the introduction of the i860 processor, Intel explored extending the i860 architecture in order to deliver high performance for other media applications, for example, image processing, texture mapping, and audio and video decompression. Several of these algorithms naturally lent themselves to SIMD processing. This effort laid the foundation for similar support for Intel's mainstream general purpose architecture, IA.

The MMX technology extension was the first major addition to the instruction set since the Intel386™ architecture. Given the large installed software base for the IA, a significant extension to the architecture required special attention to backward compatibility and design issues.

MMX technology provides benefits to the end user by improving the performance of multimedia-rich applications by a factor of 1.5x to 2x, and improving the performance of key kernels by a factor of 4x on the host processor. MMX technology also provides benefits to software vendors by enabling new multimedia-rich applications for a general purpose processor with an established customer base. Additionally, MMX technology provides an integrated software development environment for software vendors for media applications.

This paper provides insight into the process and considerations used to define the MMX technology. It also provides specifics on MMX instructions that were added to the IA as well as the approach taken to add this significant capability without adding a new software-visible architectural state.

The paper also presents application examples that show the usage and benefits of MMX instructions. Data showing the performance benefits for the applications is also presented.

Definition Process

MMX technology's definition process was an outstanding adventure for its participants, a path with many twists and turns. It was a bottom-up process. Engineering input and managerial drive made MMX technology happen.

The definition of MMX technology was guided by a clear set of priorities and goals set forth by the definition team. Priority number one was to substantially improve the performance of multimedia, communications, and emerging Internet applications. Although targeted at this market, any application that has execution constructs that fit the SIMD architecture paradigm can enjoy substantial performance speed-ups from the technology.

It was also imperative that processors with MMX technology retain backward compatibility with existing software, both operating systems and applications. The addition of MMX technology to the IA processor family had to be seamless, having no compatibility or negative performance effect on all existing IA software or operating systems. Applications that use MMX technology had to run on any existing IA operating systems without having to make any operating system modifications whatsoever and coexist in a seamless way with the existing IA application base. For example, any existing version of an operating system (i.e., Windows NT) would have to run without modifications. New applications that use MMX technology together with existing IA applications would also have to run without modifications on a processor with MMX technology.

The key principle that allowed compatibility to be maintained was that MMX technology was defined to map inside the existing IA floating-point architecture and registers [4]. Since existing operating systems and applications already knew how to deal with the IA floating-point (FP) state, mapping the MMX technology inside the floating-point architecture was a clean way to add SIMD without adding any new architectural state. The operating system does not need to know if an application is using MMX technology. Existing techniques to perform multiprocessing (sharing execution time among multiple applications by frequently switching among them) would take care of any application with MMX technology.

Another important guideline that we followed was to make it possible for application developers to easily migrate their applications to use MMX technology. Realizing that IA processors with and without MMX technology would be on the market for some time, we wanted to make sure that migration would not become a problem for software developers. By enabling a software program to detect the presence of MMX technology during run time, a software developer need develop only one version of an application that can run both on newer processors that support MMX technology and older ones which do not. When reaching a point in the execution of a program where a code sequence enhanced with MMX instructions can boost performance, the program checks to see if MMX technology is supported and executes the new code sequence. On older processors without MMX technology, a different code sequence would be executed. This calls for duplication of some key application code sequences, but our experience showed it to average less than 10% growth in program size.

We wanted to keep MMX technology simple so that it would not depend on any complex implementation which would not scale easily with future advanced microarchitecture techniques and increasing processor frequencies, thus making it a burden on the future. We made sure MMX technology would add a minimal amount of incremental die area, making it practical to incorporate MMX technology into all future Intel microprocessors.

We also wanted to keep MMX technology general enough so that it would support new algorithms or changes to existing ones. As a result, we avoided algorithm-specific solutions, sometimes sacrificing potential performance but avoiding the risk of having to support features in the future if they become redundant.

The decision of whether to add specific instructions was based on a cost-benefit analysis for a large set of existing and futuristic applications in the area of multimedia and communications. These applications included MPEG1/2 video, music synthesis, speech compression, speech recognition, image processing, 3D graphics in games, video conferencing, modem, and audio applications. The definition team also met with external software developers of emerging multimedia applications to understand what they needed from a new Intel Architecture processor to enhance their products. Applications were collected from different sources, and in some cases where no application was readily available, we developed our own. Applications we collected were broken down to reveal that, in most cases, they were built out of a few key compute-intensive routines where the application spends most of its execution time. These key routines were then analyzed in detail using advanced computer-aided profiling tools. Based on these studies, we found that key code sequences had the following common characteristics:

- Small, native data types (for example, 8-bit pixels, 16-bit audio samples)

- Regular and recurring memory access patterns
- Localized, recurring operations performed on the data
- Compute-intensive

This common behavior enabled us to come up with MMX technology, which is a solution that supports well a wide variety of applications from different domains.

Basic Concepts

Our observations of multimedia and communications applications pointed us in the direction of an architecture that would enable exploiting the parallelism noted in our studies.

Beyond the obvious performance enhancement potential gained by packing relatively small data elements (8 and 16 bits) together and operating on them in parallel, this kind of packing also naturally enables utilizing wide data paths and execution capabilities of state-of-the-art processors.

An efficient solution for media applications necessitates addressing some concepts that are fundamental to the SIMD approach and multimedia applications:

- Packed data format
- Conditional execution
- Saturating arithmetic vs. wrap-around arithmetic
- Fixed-point arithmetic
- Repositioning data elements within packed data format
- Data alignment

Packed Data Format

MMX technology defines new register formats for data representation. The key feature of multimedia applications is that the typical data size of operands is small. Most of the data operands' sizes are either a byte or a word (16 bits). Also, multimedia processing typically involves performing the same computation on a large number of adjacent data elements. These two properties lend themselves to the use of SIMD computation.

One question to answer when defining the SIMD computation model is the width or the data type for SIMD instructions. How many elements of data should we operate on in parallel? The answer depends on the characteristics of the natural organization and alignment of the data for targeted applications and design considerations. For example, for a motion estimation algorithm, data is naturally organized in 16 rows, with each row containing only 16 bytes of data. In this case, operating on more than 16 data elements at a time will require reformatting the input data. Design considerations involve issues such as the practical width of the data path and how many times functional units will replicate.

Given that current Intel processors already have 64-bit data paths (for example, floating-point data paths, as well as a data path between the integer register file and memory subsystem due to dual load/store capability in the Pentium® processor), we chose the width of MMX data types to be 64 bits.

Conditional Execution

Operating on multiple data operands using a single instruction presents an interesting issue. What happens when a computation is only done if the operand value passes some conditional check? For example, in an absolute value calculation, only if the number is already negative do we perform a 2's complement on it:

```
for   I = 1, 100
   if a[i] < 0 then b[i] = - a[i] else b[i] = a[i]
   ; Absolute value calculation
```

There are different approaches possible, and some are simpler than others. Using a branch approach does not work well for two reasons: first, a branch-based solution is slower because of the inherent branch misprediction penalty, and second, because of the need to convert packed data types to scalars.

Direct conditional execution support does not work well for the IA since it requires three independent operands (source, source/destination, and predicate vector). Keeping with the philosophy of performance and simplicity, we chose a simpler solution. The basic idea was to convert a conditional execution into a conditional assignment. Conditional assignment in turn can be implemented through different approaches. One approach would be to provide the flexibility of specifying a dynamically generated mask with an assignment instruction. Such an approach would have required defining instructions with three operands (source, source/destination, and mask). Here also, we adopted a solution that is more amenable to higher performance designs.

Compare operations in MMX technology result in a bit mask corresponding to the length of the operands. For example, a compare operation operating on packed byte operands produce byte-wide masks. These masks then can be used in conjunction with logical operations to achieve conditional assignment. Consider the following example:

```
If  True
    Ra := Rb else  Ra := Rc
```

Let us say register Rx contains all 1's if the condition is true and all 0's if the condition is false. Then we can compute Ra with the following logical expression:

Ra = (Rb AND Rx) OR (Rc ANDNOT Rx)

This approach works for operations with a register as the destination. Conditional assignment to memory can be implemented as a sequence of load, conditional assignment, and store. We rejected more efficient support for conditional stores for two reasons: first, the support requires three source operands, which does not map well to high-performance architectures, and second, the benefit of such support is dependent on support from the platform for efficient partial transfers.

The MMX instruction set contains a packed compare instruction that generates a bit mask, enabling data-dependent calculations to be executed without branch instructions and to be executed on several data elements in parallel. The bit mask result of the packed compare instruction has all 1's in elements where the relation tested for is true and all 0's otherwise (see Figure 1).

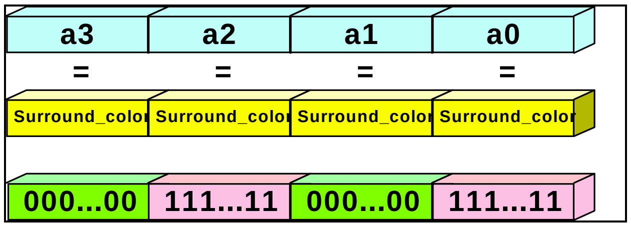

Figure 1. Packed Equal on Word Data Type

Saturating Arithmetic

Operand sizes typically used in multimedia are small (for example, 8 bits for representing a color component). An 8-bit number allows only 256 different shades of a color to be displayed. While this resolution is more than enough for what the eye can see, it presents us with a problem in computation. Given only an 8-bit representation, the accumulation of color values of a large number of pixels is likely to exceed the maximum value that can be represented by the 8-bit number. In the default computational model, if the addition of two numbers results in a value that is more than the maximum value that can be represented by the destination operand, a wrapped-around value is stored in the destination. If an application cared to safeguard against such a possibility, then it has to explicitly examine for an occurrence of an overflow.

In media applications, typically the desired behavior is to provide not the wrap-around value but the maximum value as the result. MMX technology provides an option to the application program, which determines whether a wrap-around result or maximum result is provided in case of an overflow.

There may be cases where an application wants to examine the occurrence of an overflow in a computation. Providing a flag to indicate this (i.e., indicating whether or not the value was saturated) would have been desirable. However, we decided against providing this flag, since we did not want to add any additional new states to the architecture to preserve the backward compatibility. Our analysis also showed that it was not critical to provide this information in most applications. If needed, an application can determine if saturation was encountered by comparing the result of a computation with the maximum and minimum value; typically, saturation is the correct behavior.

Fixed-Point Arithmetic

Media applications involve working on fraction values, for example, the use of a weighting coefficient in filtering averaging, etc. One way to support operations on fraction values is to provide SIMD operations for floating-point operands. However, floating-point units are hardware-intensive. Also, for several media applications, even precision of 10 to 12 binary bits and dynamic range of 4 to 6 bits are sufficient. Industry-standard floating-point (IEEE FP) requires a minimum of 23 bits of precision. Looking at application requirements and the trade-off of performance and design complexity leads to the use of a fixed-point arithmetic paradigm for several media applications. Note that some of the computations may still require the dynamic range and the precision supported by IEEE floating-point, for example, geometry transformation for state-of-the-art 3D applications.

In fixed-point computation, from the point of view of the processor architecture, computations are done on integer values, but programmer/applications interpret the integer values as fraction values. Some number of leading bits (determined by the application) are interpreted as an integer, while the remaining bits of the value are interpreted as a fraction. It is the application's responsibility to perform appropriate shifts in order to scale the number.

Repositioning of Data Elements Within Packed Data Format

The packed data format presents one other issue. There are several cases where elements of packed data may be required to be repositioned within the packed data, or the elements of two packed data operands may need to be merged. There are cases where either input or the desired output representation of a data may not be ideal for maximizing computation throughput. For example, it may be preferable to compute on color components of a pixel in "planar format" while the input may be in "packed format."

There are also situations where one needs to perform intermediate computations in wider format (perhaps packed word format), while the result is presented in packed byte format.

In the above cases, there is a need to extract some elements of a packed data type and write them into a different position in the packed result.

One general solution to this issue is to provide an instruction that takes two packed data operands and allows merging of their bytes in any arbitrary order into the destination packed data operand. However, such a general solution is expensive to implement. This solution essentially will require a full cross bar connection.

In the MMX technology architecture, we defined an instruction that requires a relatively easy swizzle network and yet allows the efficient repositioning and combining of elements from packed data operands in most cases.

The instruction *unpack* takes two packed data operands and merges them as shown in Figure 2.

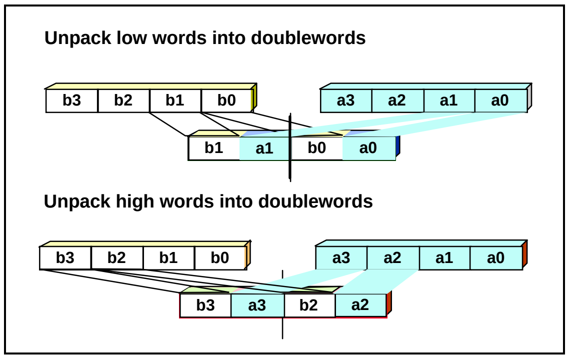
Unpack low words into doublewords
b3
b2
b1
b0
a3
a2
a1
a0
b1
a1
b0
a0
Unpack high words into doublewords
b3
b2
b1
b0
a3
a2
a1
a0
b3
a3
b2
a2

Figure 2. MMX Technology Unpacked Instruction

The *unpack* instruction can be used for a variety of efficient repositioning of data elements, including data replication, within packed data. For example, consider converting a color representation from packed form (i.e., for each pixel, four consecutive bytes represent R, G, B, and Alpha values) to planar format (i.e., four consecutive bytes represent the red component of four consecutive pixels).

Data Alignment

Use of packed data also presents data alignment issues. In some cases, the data may be aligned on its natural boundary and not on the size of the packed data operand. For example, in a motion estimation routine, the 16x16 block is aligned at an arbitrary byte boundary and not at a 64-bit boundary. Therefore, in some cases, there is a need to support efficient access of unaligned data for media applications. One approach is to support unaligned accesses directly in hardware, which generally does not work well with the high-performance cache design. Alternatively, one can limit memory accesses to aligned data and extract out the desired data from the accessed data using explicit instructions.

MMX technology includes logical shift-left and shift-right operations on 64 bits. These instructions enable using a sequence of *Shift left*, *Shift right*, and *Or* operations to assemble the desired byte from the aligned data that encompasses the desired bytes.

Features

MMX technology features include:

- New data types built by packing independent small data elements together into one register.
- An enhanced instruction set that operates on all independent data elements in a register, using a parallel SIMD fashion.
- New 64-bit MMX registers that are mapped on the IA floating-point registers.
- Full IA compatibility.

New Data Types

MMX technology introduces four new data types: three packed data types and a new 64-bit entity. Each element within the packed data types is an independent fixed-point integer. The architecture does not specify the place of the fixed point within the elements, because it is the user's responsibility to control its place within each element throughout the calculation. This adds a burden on the user, but it also leaves a large amount of flexibility to choose and change the precision of fixed-point numbers during the course of the application in order to fully control the dynamic range of values.

The following four data types are defined (see Figure 3):

• Packed byte	8 bytes packed into 64 bits
• Packed word	4 words packed into 64 bits
• Packed doubleword	2 doublewords packed into 64 bits
• Packed quadword	64 bits

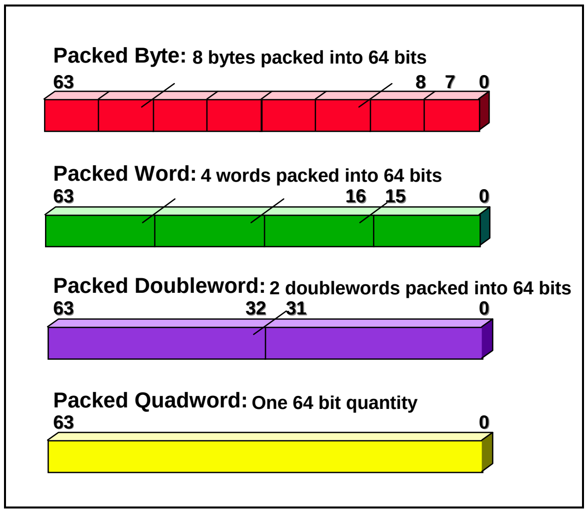
Packed Byte: 8 bytes packed into 64 bits
63
8
7
0
Packed Word: 4 words packed into 64 bits
63
16
15
0
Packed Doubleword: 2 doublewords packed into 64 bits
63
32
31
0
Packed Quadword: One 64 bit quantity
63
0

Figure 3. MMX Technology Packed Data Types

Enhanced Instruction Set

MMX technology defines a rich set of instructions that perform parallel operations on multiple data elements packed into 64 bits (8x8-bit, 4x16-bit, or 2x32-bit fixed-point integer data elements). We view the MMX technology instruction set as an extension of the basic operations one would perform on a single datum in the SIMD domain. Instructions that operate on packed bytes were defined to support frequent image operations that involve 8-bit pixels or one of the 8-bit color components of 24/32-bit pixels (Red, Green, Blue, Alpha channel). We defined full support for packed word (16-bit) data types. This is because we found 16-bit data to be a frequent data type in many multimedia algorithms (e.g., MODEM, Audio) and serves as the higher precision backup for operations on byte data. A basic instruction set is provided for packed doubleword data types to support operations that need intermediate higher precision than 16 bits and a variety of 3D graphics algorithms. Because MMX technology is a 64-bit capability, new instructions to support 64 bits were added, such as 64-bit memory moves or 64-bit logical operations.

Overall, 57 new MMX instructions were added to the Intel Architecture instruction set.

The MMX instructions vary from one another by a few characteristics. The first is the data type on which they operate. Instructions are supplied to do the same operation on different data types. There are also instructions for both signed and unsigned arithmetic.

MMX technology supports saturation on packed add, subtract, and data type conversion instructions. This facilitates a quick way to ensure that values stay within a given range, which is a frequent need in multimedia operations. In most cases, it is more important to save the execution time spent on checking if a value exceeds a certain range than worry about the inaccuracy introduced by clamping values to minimum or maximum range values. Saturation is not a mode activated by setting a control bit but is determined by the instruction itself. Some instructions have saturation as part of their operation.

MMX technology added data type conversion instructions to address the need to convert between the new data types and to enable some intermediate calculations to have more bits available for extended precision. Also, many algorithms used in multimedia and communications applications perform multiply-accumulate computations. MMX technology addressed this with a special multiply-add instruction.

MMX instructions were defined to be scalable to higher frequencies and newer advanced microarchitectures. We made them fast. All MMX instructions with the exception of the multiply instructions execute in one cycle both on the Pentium processor with MMX technology and on the Pentium® II processor. The multiply instructions have an execution latency of three cycles, but the multiply unit's pipelined design enables a new multiply instruction to start every cycle. With the appropriate software loop unrolling, a throughput of one cycle per SIMD multiply is achievable.

MMX instructions are non-privileged instructions and can be used by any software, applications, libraries, drivers, or operating systems.

Table 1 summarizes the instructions introduced by MMX technology:

Opcode	Options	Cycle Count	Description
PADD[B/W/D] PSUB[B/W/D]	Wrap-around, and saturate	1	Packed eight bytes (b), four 16-bit words (w), or two 32-bit doublewords (d) are added or subtracted in parallel.
PCMPEQ[B/W/D] PCMPGT[B/W/D]	Equal or Greater than	1	Packed eight bytes (b), four 16-bit words (w), or two 32-bit (d) elements are compared in parallel. Result is mask of 1’s if true or 0’s if false.
PMULLW PMULHW	Result is high- or low-order bits	latency: 3 throughput: 1	Packed four signed 16-bit words are multiplied in parallel. Low-order or high-order 16-bits of the 32-bit result are chosen.
PMADDWD	Word to doubleword conversion	latency: 3 throughput: 1	Packed four signed 16-bit words are multiplied and adjacent pairs of 32 results are added together, in parallel. Result is a doubleword.
PSRA[W/D] PSLL[W/D/Q] PSRL[W/D/Q]	Shift count in register or immediate	1	Packed four words, two doublewords, or the full 64-bits - quadword (q) are shifted arithmetic right, logical right and left, in parallel.
PUNPCKL[BW/WD/DQ] PUNPCKH[BW/WD/DQ]		1	Packed eight bytes (b), four 16-bit words (w), or two 32-bit doublewords (d) are merged with interleaving.
PACKSS[WB/DW]	Always saturate	1	Doublewords are packed to words or words are packed to bytes in parallel.
PLOGICALS		1	Bitwise and, or, xor, andnot.
MOV[D/Q]		1 (if data in cache)	Moves 32 or 64 bits to and from memory to MMX registers, or between MMX registers. 32-bits can be moved between MMX and integer registers.
EMMS		Varies by implementation	Empty FP register tag bits.

Table 1 lists all the MMX instructions. If an instruction supports multiple data types (byte (b), word (w), doubleword (d), or quadword (q)), the data types are listed in brackets.

64-Bit MMX Registers

MMX technology provides eight new 64-bit general purpose registers that are mapped on the floating-point registers. Each can be directly addressed within the assembly by designating the register names MM0 - MM7 in MMX instructions. MMX registers are random access registers, that is, they are not accessed via a stack model like the floating-point registers. MMX registers are used for holding MMX data only. MMX instructions that specify a memory operand use the IA integer registers to address that operand. As the MMX registers are mapped over the floating-point registers, applications that use MMX technology have 16 registers to use. Eight are the MMX registers, each 64 bits in size that hold packed data, and eight are integer registers, which can be used for different operations like addressing, loop control, or any other data manipulation. MMX data values reside in the low order 64 bits (the mantissa) of the IA 80-bit floating-point registers (see Figure 4).

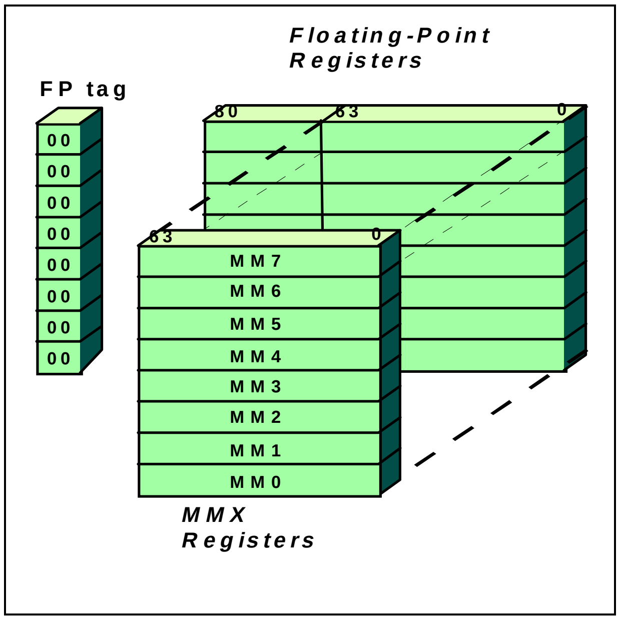

Figure 4. Mapping of MMX Registers to Floating-Point Registers

The exponent field of the corresponding floating-point register (bits 64-78) and the sign bit (bit 79) are set to ones (1’s), making the value in the register a NaN (Not a Number) or infinity when viewed as a floating-point value. This helps to reduce confusion by ensuring that an MMX data value will not look like a valid floating-point value. MMX instructions only access the low-order 64 bits of the floating-point registers and are not affected by the fact that they operate on invalid floating-point values.

The dual usage of the floating-point registers does not preclude applications from using both MMX code and floating-point code. Inside the application, the MMX code and floating-point code should be encapsulated in separate code sequences. After one sequence completes, the floating-point state is reset and the next sequence can start. The need to use floating-point data and MMX (fixed-point integer) data at the same time is infrequent. At a given time in an application, data being operated upon is usually of one type. This enabled us to use the floating-point registers to store the MMX technology values and achieve our full backward compatibility goal.

Preserving Full Backward Compatibility

One of the important requirements for MMX technology was to enable use of MMX instructions in applications without requiring any changes in the IA system software.

An additional requirement was that an application should be able to utilize performance benefits of MMX technology in a seamless fashion, i.e., it should be able to employ MMX instructions in part of the application, without requiring the whole of the application to be MMX technology-aware.

Primary backward compatibility requirements and their implications are:

- Applications using MMX instructions should work on all existing multitasking and non-multitasking operating systems.

  This requires that MMX technology should not add any new architecturally visible states or events (exceptions).

- Existing applications that do not use MMX instructions should run unchanged.

  This requires that MMX technology should not redefine the behavior of any existing IA 32-bit instructions. Only those undefined opcodes that are not relied on for causing illegal exceptions by existing software should be used to define MMX instructions.

  Also, MMX instructions should only affect the IA 32-bit state when in use.

- Existing applications should be able to utilize MMX technology without being required to make the whole application MMX technology-aware. It should be possible to employ MMX instructions within a procedure in an existing application without requiring any changes in the rest of the application.

  This requires that MMX instructions work well within the context of existing IA calling conventions for procedure calls.

- It should be possible to run an application even in an older generation of processors that does not support MMX technology.

  Using dynamically linked libraries (DLLs) for MMX and non-MMX technology processors is an easy way to do this.

- MMX instructions should be semantically compatible with other IA instructions, i.e., it should be easy to support new MMX instructions in existing assemblers. They should also have minimal impact on the instruction decoder. Another aspect of this is that MMX instructions should not require programmers to think in new ways regarding the basic behavior of instructions. For example, addressing modes and the availability of operations with memory should conceptually work the same.

The behavior of the prefix overrides should also be consistent with the IA.

No New State

The MMX technology state overlaps with the Floating-Point state. Overlapping the MMX state with the FP stack presented an interesting challenge. For performance reasons as well as for ease of implementation for some microarchitectures, we wanted to allow the accessing of the MMX registers in a flat register model. We needed to enable overlapping MMX registers with the FP stack while still allowing a flat register access model for MMX instructions. This was accomplished by enforcing a fixed relationship between the logical and physical registers for the FP stack, when accessed via MMX instructions. Additionally, every MMX instruction makes the whole MMX register file valid. This is different from the floating-point stack model, where new stack entries are made valid only if the instruction specifies a "push" operation.

MMX instructions themselves do not update FP instruction state registers (for example, FP opcode, FOP, FP Data selector, FDS, FP IP, FIP, etc.). The FP instruction state is used only by FP exception handlers. Since MMX instructions do not create any computation exceptions, this state is really not meaningful for MMX instructions. Additionally, not updating these states eliminates the complexity of maintaining this state for MMX technology implementations. Therefore, we made a decision to let the FP instruction state register point to the last FP instruction executed even though future MMX instructions will update the FP stack and TAG register. Eventually, when an FP instruction is executed, all of the FP instruction state gets updated. Therefore, FP exception handlers always see consistent FP instruction state.

No New Exceptions

MMX instructions can be viewed as new non-IEEE floating-point instructions that do not generate computation exceptions. However, similar to FP instructions, they do report any pending FP exceptions. For compatibility with existing software, it is critical that any pending FP exception is reported to the software prior to execution of any MMX instruction which could update the FP state.

At the point of raising the pending FP exception, the FP exception state still points to the last FP instruction creating the FP condition. Therefore, the fact that the exception gets reported by an MMX instruction instead of an FP instruction is transparent to the FP exception handler.

Additional exceptions that are pertinent to MMX technology are memory exceptions, device-not-available (DNA - INT7) exceptions, and FP emulation exceptions.

Handling of memory exceptions, in general, does not depend on the opcode of the instruction causing the exception. Therefore, MMX technology exceptions do not cause a malfunction of any memory access-related exception handler. Our extensive compatibility verification validated this further.

A DNA exception is caused when the TS bit in CR0 is set, and any other instruction that could modify the FP state is issued. This includes execution of an MMX instruction when the TS bit is set. In this case, similar to the FP case, a DNA exception is invoked. The response of this exception is to save the FP state and free it up for use by future FP/MMX instructions. This exception handler also does not have a use for the opcode of the instruction causing this exception.

When the CR0.EM bit is set, a floating-point instruction causes an FP emulation exception. In this case, instead of using FP hardware, FP functionality is supported via software emulation. Since the MMX technology architecture state overlaps with the FP architecture state, the issue arises as to the correct behavior for MMX instructions when the CR0.EM bit is set.

Causing an emulation exception for MMX instructions when CR0.EM is set is not the right behavior since the existing FP emulator does not know about MMX instructions. Therefore, the first natural choice seemed to ignore CR0.EM for MMX technology. However, this choice has a problem. Ignoring CR0.EM for MMX instructions would result in two separate contexts for the FP Stack and TAG words: one context in the emulator memory for FP and one context in the hardware for MMX instructions. This leads to an architectural inconsistency between the cases when CR0.EM is set and when it is not set.

We had to find some other logical way to deal with this without defining any new exceptions. We chose to define the CR0.EM = 1 case to result in an illegal opcode exception. Thus, essentially when CR0.EM is set, the MMX technology architecture extension is disabled.

Choice of Opcodes for MMX Instructions

The MMX instruction opcodes were chosen after extensive analysis of the undefined opcode map. We had to make sure that the available opcodes were really unused. This required ensuring that no software was relying on the illegal opcode fault behavior of these opcodes. Intel was already working with software vendors to ensure that they relied only on one specific encoding

0FFF to cause an illegal opcode fault. Other encoding may cause an illegal exception fault in future implementations.

Except for a few cases, we found that software was using only prescribed encoding for causing a program-controlled invalid opcode fault.

Only address prefixes are defined to be meaningful for MMX instructions. Use of a Repeat, Lock, or Data prefix is illegal for MMX instructions. The address prefix has the same behavior as for any other instruction.

Use of FP DLL Model for MMX Code

To enable common multimedia applications for processors with and without MMX technology, we chose to promote the Dynamic Linked Library (DLL) model as the primary model to support MMX instructions.

In the DLL model, depending upon whether the processor provides MMX technology support in hardware (the processor CPUID provides this information), the appropriate version of the media library function is linked dynamically.

MMX technology DLLs suggest the same guidelines as that of FP DLLs. The primary guidelines are:

- At the end of a DLL, leave the floating-point registers in the correct state for the calling procedure. This generally means leaving the floating-point stack empty, unless a procedure has a return value. This also means that the caller should check for, and handle, any FP exceptions that it might have generated. Essentially, the callee should not see an exception invocation due to an exception result generated by the caller.

- Do not assume that the floating-point state remains the same across procedures. The callee can typically assume that at entry, the FP stack is empty unless there is some set convention for parameter passing.

Note that nothing in the MMX technology architecture depends on these guidelines for functional correctness. MMX technology can be used in any other usage models.

MMX technology provides an instruction to clear all of FP state with a single instruction (EMMS instruction). If some DLL is written to return with the FP stack only partially empty, one needs to use a combination of EMMS and floating-point loads to create the correct FP stack state. Clean the state of MMX with EMMS instruction.

Performance Advantage

We will analyze the performance enhancement due to MMX technology through an example of a matrix-vector multiplication very much like the one in Figure 5. The multiply-accumulate (MAC) operation is one of the most frequent operations in multimedia and communications applications used in basic mathematical primitives like matrix multiply and filters.

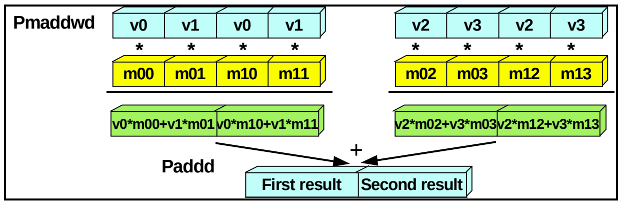

Figure 5. MMX Technology Matrix-Vector Multiplication

A multiply-accumulate operation (MAC) is defined as the product of two operands added to a third operand (the accumulator). This operation requires two loads (operands of the multiplication operation), a multiply, and an add (to the accumulator). MMX technology does not support three operand instructions; therefore, it does not have a full MAC capability. On the other hand, the packed multiply-add instruction (PMADDWD) is defined, which computes four 16-bit x 16-bit multiplies generating four 32-bit products and does two 32-bit adds (out of the four needed). A separate packed add doubleword (PADDD) adds the two 32-bit results of the packed multiply-add to another MMX register, which is used as an accumulator.

For this performance example, we will assume both input vectors to be the length of 16 elements, each element in the vectors being signed 16 bits. Accumulation will be performed in 32-bit precision. The Pentium processor, for example, would have to process each of the operations one at a time in a sequential fashion. This amounts to 32 loads, 16 multiplies, and 15 additions, a total of 63 instructions. Assuming we perform 4 MACs (out of the 16) per iteration, we need to add 12 instructions for loop control (3 instructions per iteration, increment, compare, branch), and one instruction for storing the result. 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 (utilizing both pipelines) takes just over 200 cycles on a Pentium processor microarchitecture. The cycle count is dominated by the integer multiply being a non-pipelined 11-cycle operation. Under the same conditions but assuming the data is in a floating-point format, the floating-point optimized assembly version executes in 74 cycles. The floating-point version is faster (assuming the data is in floating-pointing format) since the floating-point multiply takes three cycles to execute and is a pipelined unit.

MMX technology, on the other hand, 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 any loop control instructions. This is because four PMADDWDs already perform all the 16 required MACs. The MMX instruction count is four times less than when using integer or floating-point operations! With the same assumptions as above on a Pentium processor with MMX technology, an MMX technology-optimized assembly version of the code utilizing both pipelines will execute in only 12 cycles.

Continuing the above example, assume a 16x16 matrix is multiplied by a 16-element vector. This operation is built of 16 Vector-Dot-Products (VDP) of length 16. Repeating the same exercise as before and assuming a loop unrolling that performs four VDPs each iteration, the regular Pentium processor code will total 4*(4*76+3) = 1228 instructions. Using MMX technology will require 4*(4*19+3) = 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 1200 cycles to complete in comparison to 207 cycles for the MMX code version.

Intel has introduced two processor families with MMX technology: the Pentium processor with MMX technology and the Pentium II processor. The performance of both processors was compared on the Intel Media Benchmark (IMB) [5,6], which measures the performance of processors running algorithms found in multimedia applications. The IMB incorporates audio and video playback, image processing, wave sample rate conversion, and 3D geometry. Figure 6 and Table 2 compare the Pentium processor with MMX technology and the Pentium II processor against the Pentium processor and the Pentium® Pro processor.

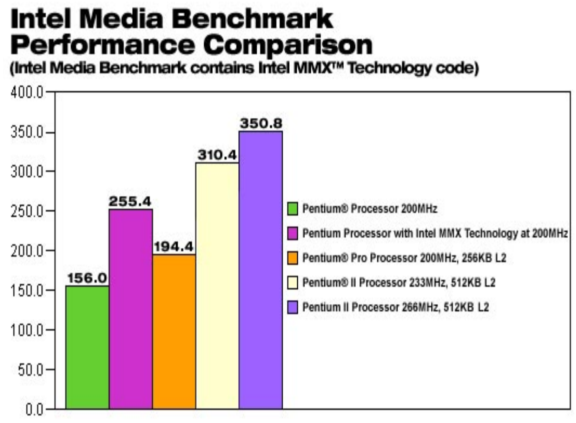

Figure 6. Intel Media Benchmark Performance Comparison

Intel Media Benchmark

Performance Comparison

	Pentium processor 200MHz	Pentium processor 200MHz—MMX Technology	Pentium Pro processor 200MHz—256KB L2	Pentium II processor 233MHz—512KB L2	Pentium II processor 266MHz—512Kb L2
Overall	**156.00**	**255.43**	**194.38**	**310.40**	**350.77**
Video	155.52	268.70	158.34	271.98	307.24
Image Processing	159.03	743.92	220.75	1,026.55	1,129.01
3D Geometry*	161.52	166.44	209.24	247.68	281.61
Audio	149.80	318.90	240.82	395.79	446.72

Pentium processor and Pentium processor with MMX technology are measured with 512K L2 cache

* No MMX™ technology code

Table 2. Intel Media Benchmark Performance Comparison - Breakdown Per Application

Summary

MMX technology implements a high-performance technique that enhances the performance of Intel Architecture microprocessors for media applications. The core algorithms in these applications are compute-intensive. These algorithms perform operations on a large amount of data, use small data types, and provide many opportunities for parallelism. These algorithms are a natural fit for SIMD architecture. MMX technology defines a general purpose and easy-to-implement set of primitives to operate on packed data types.

MMX technology, while delivering performance boost to media applications, is fully compatible with the existing application and operating system base.

MMX technology is general by design and can be applied to a variety of software media problems. Some examples of this variety were described in this paper. Future media-related software technologies for use on the Intranet and Internet should benefit from MMX technology.

Pentium processors with MMX technology provide a significant performance boost (approximately 4x for some of the kernels) for media applications.Performance gains from the technology will scale well with an increased processor operating frequency and future microarchitectures.

Acknowledgment/References/Authors

Acknowledgment

We would like to thank a large number of architects and software developers in Intel for their help and support in the definition process and application analysis. Other members of the core MMX technology definition team were David Bistry, Robert Dreyer, Carole Dulong, Steve Fischer, Andy Glew, Eiichi Kowashi, Koby Gottlieb, Larry Mennemeier, Mike Keith, and Benny Eitan.

References

[1] A. Peleg, U. Weiser, *MMX™ Technology Extension to the Intel Architecture*, IEEE Micro, Vol. 16, No. 4, August 1996, pp. 42-50.

[2] A. Peleg, S. Wilkie, U. Weiser, *Intel MMX for Multimedia PCs*, Communications of the ACM, Vol. 40, No. 1, January 1997, pp. 25-38.

[3] Intel Corporate Literature, *i860™ Microprocessor Family Programmers* Reference *Manual*, Order number 240875. Intel Corporate Literature Sales, 1991.

[4] *Pentium® Family User's Manual, Volume 3: Architecture and Programming Manual,* Order number 241430, Intel Corporate Literature Sales, Mt. Prospect, IL, 1994.

[5] M. Slater, *The Land Beyond Benchmarks, Computer and Communications OEM Magazine*, Vol. 4, No. 31, September 1996, pp. 64-77.

[6] Intel Media Benchmark URL: http://pentium.intel.com/procs/perf/icomp/imb.htm

Authors

Millind Mittal is a Staff Computer Architect with the Microprocessors Division at Intel. He focuses on emerging architecture issues for Intel processors.

Millind was one of the primary architects of the initial extension of i860 to include SIMD instructions. Over the years, he has led and participated in several architecture definitions, including parts of IA-64 architecture. He was a member of the MMX technology definition team, with primary focus on the software model. He has also worked as a microarchitect for processor projects, and led a team performing processor microarchitecture research.

Millind received his B. Tech. in EE from the Indian Institute of Technology in Kanpur, India in 1983 and a MS in Computer Systems Engineering from RPI in 1985. His email address is mmittal@ccm.sc.intel.com.

Alex Peleg is a Staff Computer Architect within the Israel Design Center Architecture group for Intel's operations at Haifa, Israel. He is responsible for a team of architects dealing with the definition of architectures for Intel's CPUs.

Alex joined the Intel Israel Haifa Design Center in 1991. His initial role was as a computer architect working on the graphics architecture of Intel's i860 processor as well as research into multimedia support on future IA processors. He then led parts of the MMX technology definition team. He was also instrumental in evaluation of the performance benefits of the MMX technology for different Intel processors.

Alex received his BSCS and MSEE degrees from the Israel Institute of Technology—the Technion (1989, 1991). His email address is apeleg@iil.intel.com.

Uri Weiser received his BSEE and MSEE degrees from the Technion (1970, 1975) and his Ph.D. CS from the University of Utah (1981).

Uri joined the Israeli DOD in 1970 to work on super-fast analog feedback amplifiers. Later Uri worked at National Semiconductor where he led the design of the NS32532 microprocessor. Since 1988, Uri has been with Intel, leading various architecture activities such as Pentium processor feasibility studies, IA roadmaps, Intel's MMX technology architecture definition, and IA microarchitecture research.

Uri is an Intel Fellow and is the Director of Computer Architecture in Intel's Development Center in Haifa, Israel. Uri also holds an adjunct Senior Lecturer position at the Technion and is an Associate Editor of the IEEE Micro magazine. His email address is uri_weiser@ ccm.idc.intel.com.