Intel MMX for Multimedia PCs

Accelerating multimedia and communications applications, especially on the Internet, MMX technology, or multimedia extensions, for Intel microprocessors—starting with the Pentium—features 57 new instructions and other architectural enhancements to boost system performance.

Alex Peleg, Sam Wilkie, and Uri Weiser

THE VOLUME AND COMPLEXITY OF DATA processed by today's PC is increasing exponentially, placing incredible demands on microprocessor performance. The potential of the Internet, games, "edutainment" applications, interactive video, 3D graphics, animation, audio, and virtual reality—all of which demand ever-increasing performance—motivated Intel to develop MMX technology. MMX technology improves the performance of current and future graphics and communications applications while maintaining compatibility with the existing Intel Architecture (IA) software base of applications and operating systems.

MMX technology is an extension of IA and is IA's most significant enhancement since the Intel 386 processor, which in 1985 extended the architecture to 32 bits. MMX technology includes new instructions and data types to achieve increased levels of performance on the host CPU by exploiting the parallelism inherent in many of the algorithms in these applications. MMX can deliver 50%–100% performance gains for multimedia and communications applications over the same applications run on the same processor without

MMX technology. (Intel's media benchmark shows a 65% gain from MMX technology.) MMX technology also enables new applications (e.g., MPEG2 full-motion-video software-only decoding) while speeding the CPU to execute existing applications with more time freed for other tasks.

The new technology was designed so the performance gains scale well with processor operating frequencies and future architecture generations. It will be integrated first into the Pentium and P6 family processors, giving these processors an extra boost in capability, and will also appear on all future IA processors.

MMX Technology Concepts

We observed that MMX technology's potential target applications share several characteristics:

- Small native data types (e.g., 8-bit pixels, 16-bit audio samples)
- Compute-intensive recurring operations performed on these data types
- A lot of inherent parallelism.

These characteristics pointed the MMX technology defini-

tion team in the direction of a single-instruction-multiple-data (SIMD) architecture in which one instruction performs the same operation on multiple data elements in parallel. This parallel operation on relatively small data elements (8 and 16 bits) is the fundamental factor behind the MMX technology performance boost. The benefits of a SIMD extension have also been identified by other processor architectures and appear in, for example, the Sun Microsystems SPARC-Visual Instruction Set (VIS) [10] and in the Hewlett-Packard PA-RISC 2.0 MAX-2 instruction set [5].

UP TO NOW, WHEN PROCESSING 8- OR 16-BIT data, the existing 32- or 64-bit CPU bandwidth and processing resources in Intel processors were underutilized. Only the low-order 8 or 16 bits were manipulated, leaving the remaining bits "unemployed." MMX technology processing of independent small data elements together enables full utilization of the wide processing capabilities of the CPU. We chose a data width of 64 bits because our studies indicated that using 64 bits of packed elements would enable a substantial performance boost above the regular operation of processing one datum at a time. The 64 bits of packed elements were also a practical idea, since the Pentium and P6 generations of processors use 64-bit-wide data buses, as opposed to the 32-bit-wide buses on previous generations of processors.

MMX technology was defined with clear guidelines and specific goals in mind. Foremost, it had to improve substantially the performance of multimedia, communications, and other numeric-intensive applications. Another important principle guiding definition of the extension was to keep it independent of current microarchitectures, so MMX technology would scale easily with future advanced microarchitecture techniques and higher processor frequencies in future Intel processors.

It was also imperative that processors with MMX technology retain backward compatibility with existing software, including operating systems and applications. All existing IA software and IA operating systems had to run without modification on new processors that support MMX technology and in the presence of new applications that use MMX technology. For example, any existing version of the Windows operating system (e.g., Windows 95) can run without modification on a processor with MMX technology.

We also had to ensure coexistence of existing applications and new applications using MMX technology.

Advanced operating systems enable multiple programs (called *tasks* in the jargon of operating-system developers) to seemingly run in parallel by time sharing the CPU among them. This time sharing is called multitasking. New applications using MMX instructions should be able to multi-task with any other applications. This requirement placed constraints on the MMX technology definition. Therefore, we could not create a new MMX mode or state (e.g., new registers) because we would have needed to have operating systems modified to take care of the new additions. The main technique for maintaining full compatibility of MMX technology was "hiding" it inside the existing floating-point state and registers (current operating systems and applications are designed to work with the floating-point state). An operating system does not need to know whether MMX technology is present, since it is hidden inside the floating-point state [1]. Applications check for the presence
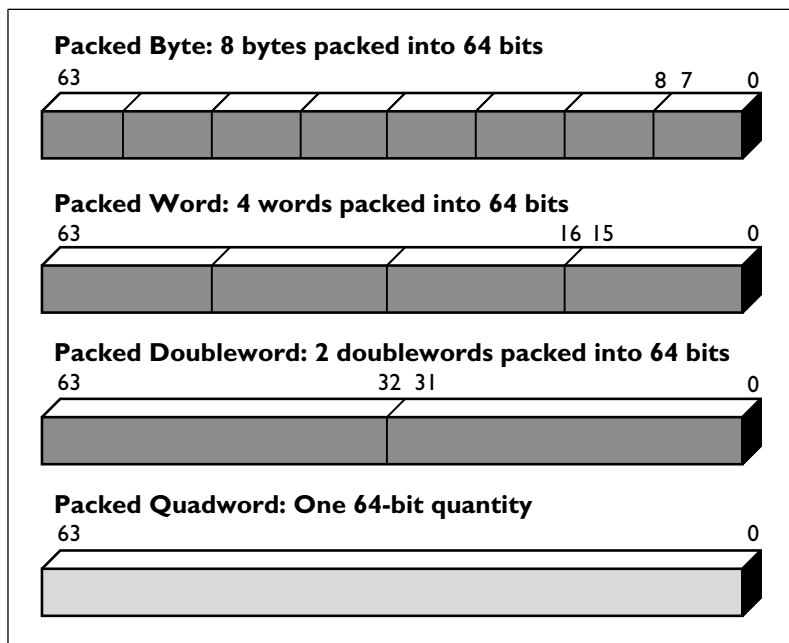

Figure 1. MMX technology data types

of MMX technology; if it is built into the processor, they use the new instructions.

MMX technology's definition process proved an unusual adventure. It was engineering input, not managerial drive, that led the way. A group of architects and software engineers analyzed the potential performance of existing and futuristic applications, including graphics, MPEG video, music synthesis, speech compression, speech recognition, image processing, 3D graphics in games, video conferencing, modems, and audio. The engineers and architects met with external software developers to learn what they needed from a new IA processor to enhance their multimedia and communications applications. The applications were analyzed to

Opcode	Options	Description
Padd[b/w/d] Psub[b/w/d]	Wrap around and saturate	Parallel Add and Subtract of packed eight bytes, four 16-bit words, or two 32-bit doublewords.
Pcmpeq[b/w/d] Pcmpgt[b/w/d]	Equal or greater than	Parallel Compare of eight bytes, four 16-bit words, or two 32-bit doublewords. Result is mask of 1s if true or 0s if false.
Pmullw Pmulhw	Result is high or low order bits	Parallel Multiply of four signed 16-bit words. Low-order or high-order 16-bits of the 32-bit result are chosen.
Pmaddwd	Word to double word conversion	Parallel Multiply-Add of four signed 16-bit words. Adjacent pairs of 32-bit results are added together. Result is a doubleword.
Psra[w/d] Psll[w/d/q] Psrl[w/d/q]	Shift count in register or immediate	Parallel Shift of 4 words, 2 doublewords, or the full 64 bits are shifted arithmetic right, logical right and left.
Punpckl[bw/wd/dq] Punpckh[bw/wd/dq]		Parallel Unpacking (interleaved merge) of eight bytes, four 16-bit words, or two 32-bit doublewords.
Packss[wb/dw]	Always saturate	Parallel Packing of doublewords to words or words to bytes.
Pand Pandn Por Pxor		64-bit bitwise logical operations
Mov[d/q]		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		Empty FP registers tag bits.

Table 1. Summary of the MMX instruction set. (If an instruction supports multiple data types—byte [b], word [w], doubleword [d], or quadword [q]—the data types are in brackets.)

MMX Technology Data Structures and Enhanced Instruction Set

Many multimedia algorithms execute the same instructions on many pieces of data in a large data set. Standard processors process only one piece of data with each instruction. MMX technology processes several pieces of data with each instruction—a simple type of parallelism that provides a big performance boost for many multimedia algorithms. Typical elements of data are usually small: 8 bits per element for pixels or 8 bits for each pixel color component— red, green, and blue—used in graphics and video; 16 bits per element for audio samples or as a higher-precision backup for 8-bit operations; and 32 bits per element for general computing and for some 3D graphics algorithms. (Table 1 summarizes the types of instructions included in MMX and the operations they perform.)

As a result, MMX technology defines new data types, which are 64 bits in total size and are composed of independent smaller-size data elements. Thus, we called them "packed data types." Each element within a packed data type is a fixed-point integer. The programmer controls the place of the fixed point within each element and is responsible for its placement throughout the calculation. While this control means an extra burden for programmers, it also gives them a large amount of flexibility to choose and change fixed-point formats during the application course to fully control the dynamic range of their values.

Four data types are defined in MMX technology (see Figure 1):

- **Packed byte:** Eight bytes packed into one 64-bit quantity
- **Packed word:** Four words packed into one 64-bit quantity
- **Packed doubleword:** Two doublewords packed into one 64-bit quantity
- **Quadword:** One 64-bit quantity

A rich set of MMX instructions are defined to perform the parallel operations on multiple data elements packed into the new 64-bit data types (8×8-bit, 4×16-bit, or

identify the most compute-intensive routines, which were then analyzed in detail using advanced computer-aided engineering tools. These studies and the performance potential they showed convinced Intel of the need to integrate the new technology as soon as possible and to fully convert all IA processors to include MMX technology.

Main Features

Here we introduce the main features of MMX technology and its interesting new instructions [7] using simple examples as a guide. The examples also show how the technology and its instructions are used to boost the performance in different kinds of multimedia algorithms and applications.

MMX technology processes several pieces of data with each instruction—a simple type of parallelism that provides a big performance boost for many multimedia algorithms.

2×32-bit fixed-point data elements). MMX technology extends the basic integer instructions into SIMD versions. These instructions include `add`, `subtract`, `multiply`, `compare`, and `shift`. MMX technology also added data-type conversion instructions to address the need to convert between the new data types. As MMX technology is a 64-bit capability, new instructions to support 64-bit operations were added, including 64-bit memory moves and 64-bit logical operations. And because many algorithms used in multimedia and communications applications perform `multiply-accumulate` computations, MMX includes a special `multiply-add` instruction.

For packed data types, MMX technology provides its most complete instruction support for packed-word (16-bit) data types, because we found 16-bit data to be the most general and useful for the wide category of multimedia algorithms. Such support also serves as the higher-precision backup for operations on byte data. The packed-byte data type is supported with the same instructions as the packed 16-bit word, with the exception of `multiply`, which is generally better done in the larger data types. Basic support is provided for packed-doubleword data types to support operations that need higher precision than 16 bits (e.g., `multiply-accumulate`) and a variety of 3D graphics algorithms. Overall, 57 MMX instructions were added to the IA.

The MMX instructions vary from one another by a few characteristics. For example, different instructions are supplied to do the same operation on different data types; one instruction operates on a packed-byte and another operates on a packed-word data type. Some instructions also vary as to whether they treat values on which they operate as signed or unsigned.

A major feature of MMX instructions is saturation arithmetic [2], which is important to many graphics routines. For example, if two medium-blue pixels are added together, saturating arithmetic ensures the result is a dark blue or black. Saturation arithmetic is certainty different from standard integer math, which can add two medium-blue pixels and in some cases produce a light-colored result.

In regular arithmetic, when an operation overflows or underflows, the most significant bit is usually truncated. Truncating the most significant bit is sometimes called "wrap-around" arithmetic because the effect of truncating the overflow or underflow bit can cause, for example, the result of adding two large numbers to be smaller than any of the input operands. Adding the 16-bit unsigned integer numbers F000h (hexadecimal representation) and 3000h generates a 17-bit result, or 12000h. But because the result size is limited to 16 bits, the high-order bit gets truncated and the actual result is 2000h (see Figure 2a), smaller than either input operands. Saturating arithmetic avoids this effect, so the instruction result, instead of wrapping around,
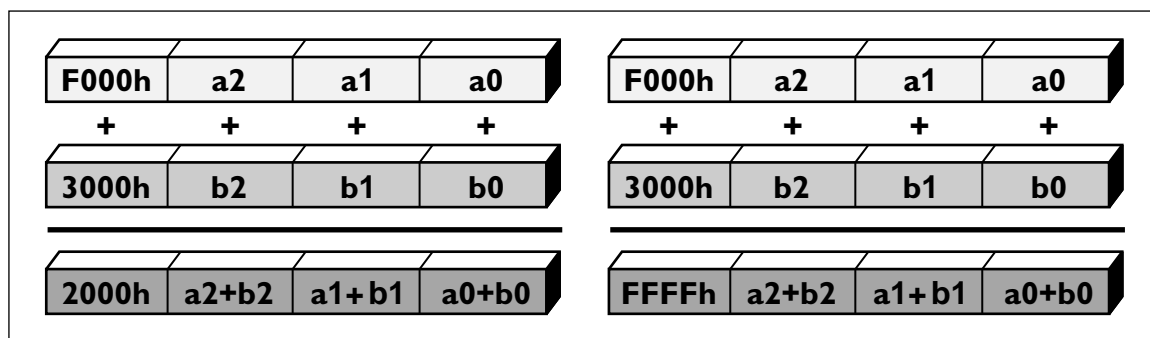
Figure 2. Saturating arithmetic

Figure 2a. Wrap-around packed adds on word data type

Figure 2b. Saturating packed adds on word data type

becomes the largest or smallest possible representable number in the data type of the operation. In the case of unsigned 16-bit integer numbers, the largest number would be FFFFh. Thus, the result F000h + 3000h is FFFFh (see Figure 2b).

MMX technology supports two types of saturating arithmetic: signed saturation and unsigned saturation. In signed saturation, the operands and results of the operation are considered signed numbers. For example, in the `packed word` data type, the largest possible value is $(2exp(n-1) - 1)$; the smallest possible value is $(-2exp(n-1))$, where n is the number of bits available. In unsigned arithmetic, the largest possible value is $2exp(n) -1$; the smallest possible value is 0.

The parallelism and saturating arithmetic in MMX technology are useful in some video conferencing compression schemes [4]. Instead of directly encoding each frame in a video sequence,

Figure 3. Image composting on color plane representation

it is better to first calculate the differences between the current frame and a recent previous frame. If the two frames are very similar (usually the case in video), it is easy to see that the resulting difference frame can be represented with less information than the original frame. So, for all the pixels in the frames, a pixel-to-pixel difference is computed. The neat thing about this operation is that all the pixel differences can be computed in parallel, because they are all independent operations. The problem is that subtracting two 8-bit unsigned pixels may result in a 9-bit negative number. Saturating arithmetic can be used in a simple way to eliminate any negative numbers during the absolute-difference calculation. The technique uses unsigned saturating subtraction to subtract pixel A from pixel B; then the reverse, subtract-
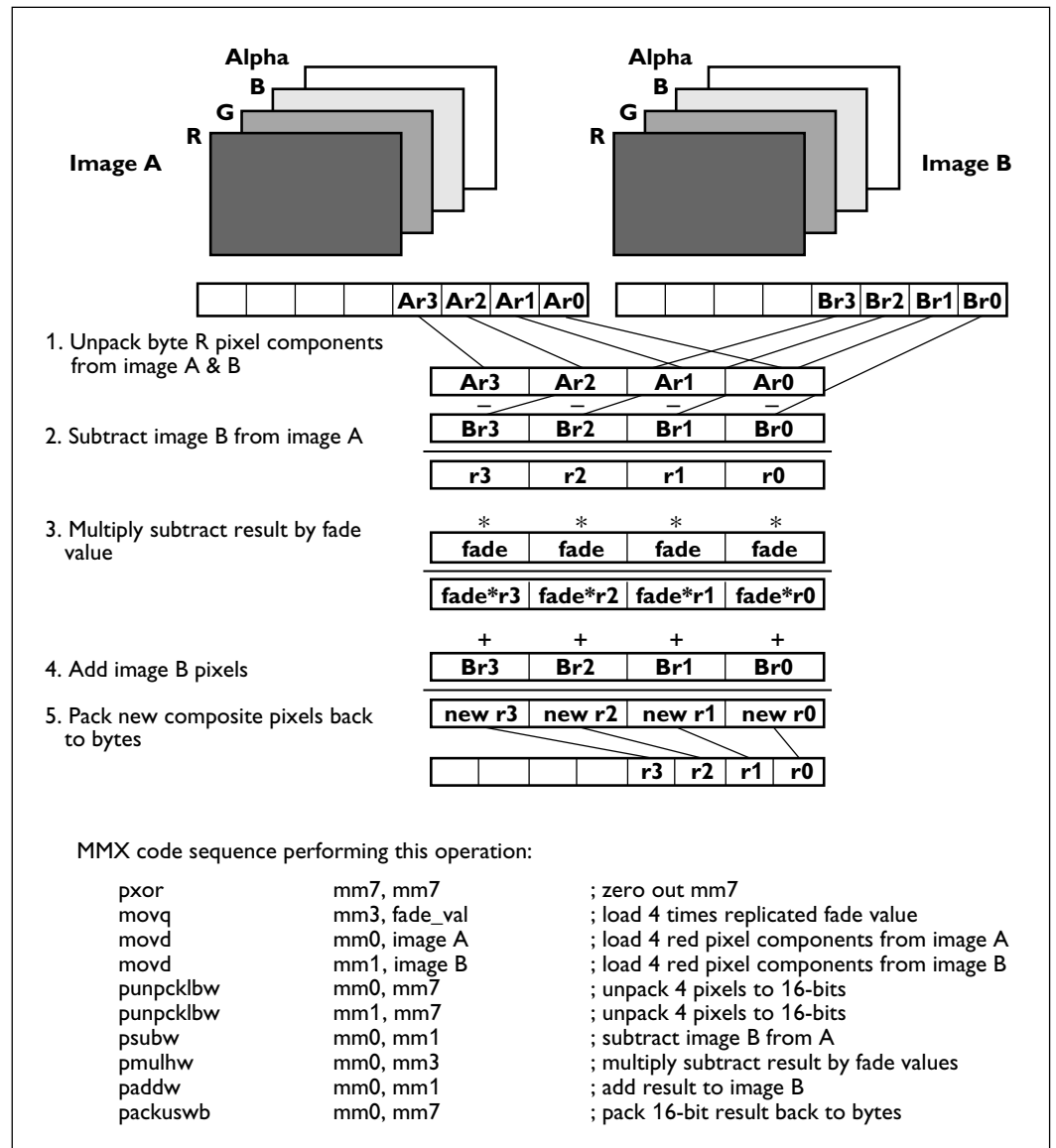
ing pixel B from pixel A. One of these differences is a positive number, the other a negative number that will be clamped to 0 because the instruction is the saturating type. We don't know which difference is positive and which is the 0 (the negative number clamped). So, after these two saturating subtractions, the results are combined using a logic OR operation. This operation can be done in parallel on 8 bytes at a time, providing great performance for this much-used operation.

Saturating arithmetic is also useful in traditional graphics. For example, Gouraud shading is a standard way to render 3D images so they look more realistic [11]. In this technique, polygons are shaded by interpolating color values across scan lines during rendering. Somewhere along a scan

line, calculations may start to overflow. Unless precautions are taken, overflow may occur and generate (as a result of the wrap-around effect) a completely different value from the one expected. A dark polygon being shaded toward black may suddenly start having white pixels. Saturation makes sure these problems do not occur because results clamp to the maximum black value and do not overflow to white.

Exploiting Different Kinds of Parallelism

There are often several ways of exploiting the data parallelism in a particular algorithm. The choice for the most efficient use of MMX instructions should be driven by how the data is laid out in memory or whether the programmer can change the way the data flows through an algorithm.

Image compositing is a set of techniques to combine two (or more) images to create special effects. A popular example is the fade-in-fade-out effect in video production. The computation performed between images A and B is a weighted average, represented as:

```
A*fade + B*(1-fade) = fade * (A-B) + B
```

Gradually changing the fade value from 1 to 0 across a few video frames generates the fade-in-fade-out effect. Before looking at how MMX technology exploits the parallelism in this operation, it is necessary to understand how image data is stored in memory. There are two different formats. Both are popular, although one is better for parallel processing and for MMX technology. In the first format, often called "planer," the information for each image is

Figure 4. Packed equal on word data type

stored in memory per color plane. In other words, all the red (R) components are at successive addresses in memory; all the green (G) components are at successive addresses; and all the blue (B) components are at successive addresses. All components of each color plane of Image A have to be multiplied by the corresponding fade value. With all of the R color components at successive addresses, it is easy for

MMX to grab multiple elements of the R plane in a single memory access and multiply them in parallel against the fade value (replicated the same number of times as the color elements). This sequence is the best format for parallelism. The same operation is used on all the data in the color plan. MMX technology simply loads that data, executes the `multiply` and writes out the data. In this example, the 8-bit pixel components are converted to 16-bit elements before the operation to accommodate MMX's 16-bit `multiply` capability (see Figure 3).

THE SECOND FORMAT, OFTEN CALLED "CHUNKY," stores the image as a series of complete color pixels 32 bits wide and composed of four 8-bit components: R, G, B color components and an alpha component (usually used as a transparency value of the pixel). MMX technology can process two of these complete pixels simultaneously but wastes the operations on alpha components. This pixel representation means 16 bits out of the 64 bits in the register are wasted and the total number of operations needs to be increased to process the data as complete pixels (also called "chunks").

Data-Dependent Computations

Multimedia algorithms usually exhibit data-independent control flow, meaning each operation can execute without needing to know the results of a previous operation. These algorithms are the most straightforward for the new technology to optimize. But some important algorithms need to know the results of a previous operation before proceeding. Such algorithms need to make use of logical operations to fit into MMX technology. An example is overlaying a *sprite* over a graphic (e.g., the mouse pointer on your computer screen overlaid on top of the word processing screen). A sprite is a separate image encompassed within a 2D array. The rest of the array is filled up by a value interpreted to be the "clear" color that does not appear when the sprite is overlaid. Overlaying the sprite on graphics or video involves checking each pixel taken from the sprite array to make sure it is not the clear color surrounding it. If a specific pixel in the sprite array does not match the clear color, the software knows it is really a pixel from the sprite image itself and writes it to the output frame. On the other hand, if the pixel matches the clear color, the software knows it is not a sprite pixel, and instead of writing it to the output frame, writes out the respective pixel from the scene being overlaid. The operation we want to perform when overlaying a sprite in array(A) over scene(C) is:
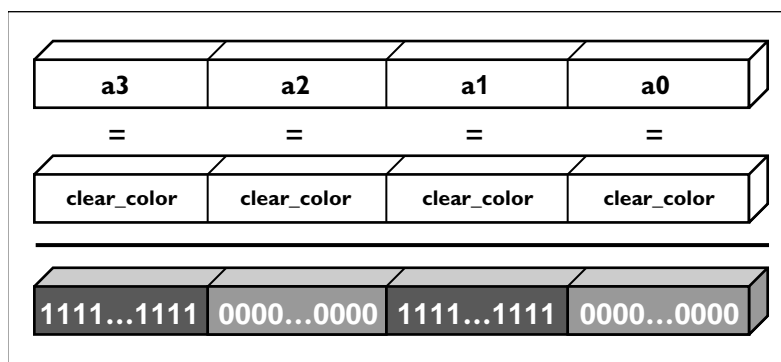
Figure 5 content:

Phase 1

a3	a2	a1	a0
=	=	=	=
clear_color	clear_color	clear_color	clear_color

1111…1111	0000…0000	1111…1111	0000…0000

Phase 2

a3	a2	a1	a0

A and (Complement of Mask)

0000…0000	1111…1111	0000…0000	1111…1111

c3	c2	c1	c0

C and Mask

1111…1111	0000…0000	1111…1111	0000…0000

0	a2	0	a0

c3	0	c1	0

**OR the two results
to finish the overlay**

c3	a2	c1	a0

MMX code sequence performing this operation:

```
movq      mm0, sprite_array    ; load pixels from sprite array
movq      mm2, mm0             ; make copy of sprite array pixels
movq      mm4, scene           ; load pixels from scene
movq      mm1, clear_color     ; load prereplicated clear-color values
pcmpeqw   mm0, mm1             ; compare sprite array pixels to clear color
pand      mm4, mm0             ; select scene pixels
pandn     mm0, mm2             ; select sprite pixels
por       mm0, mm4             ; generate overlay
```
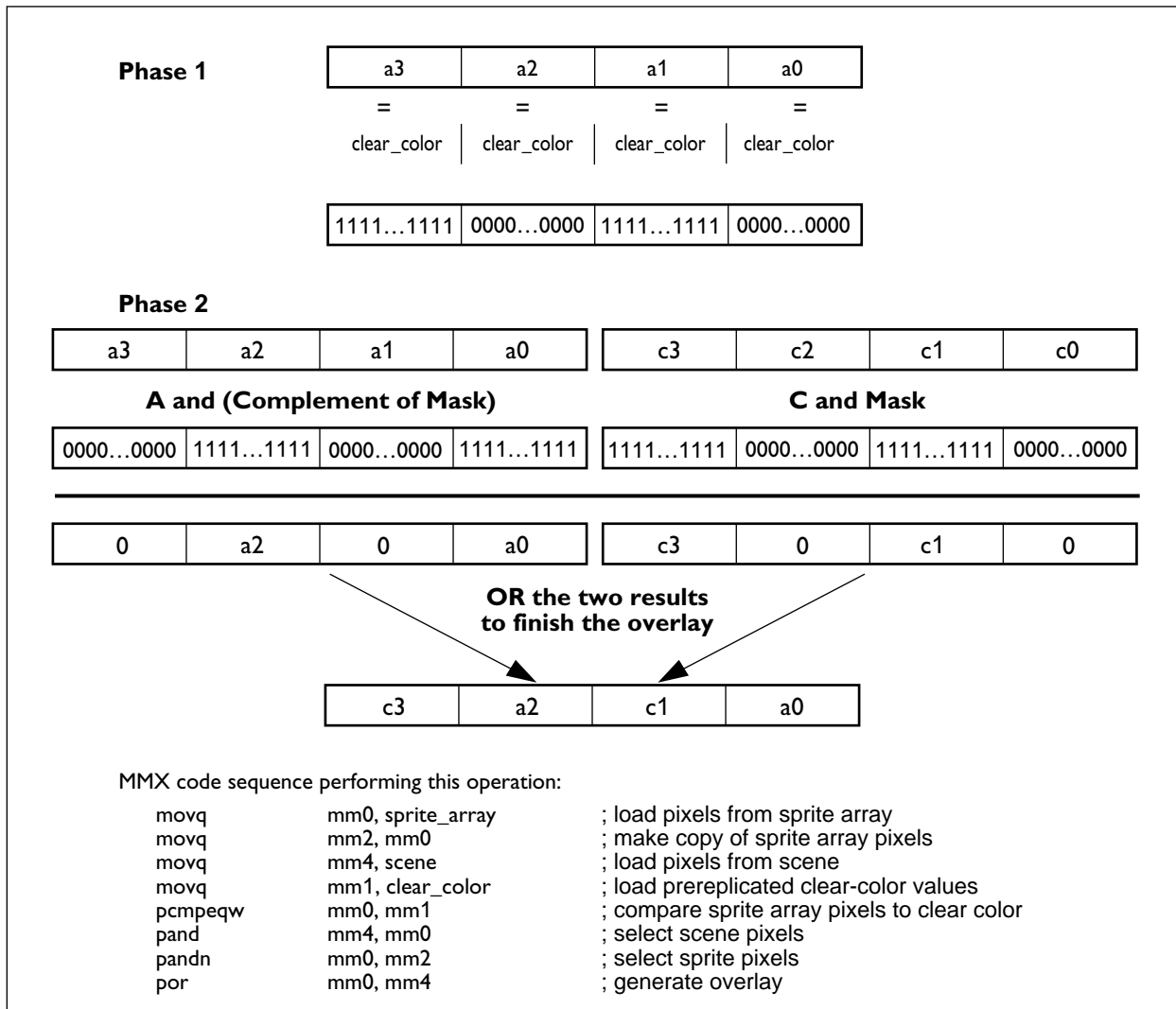
Figure 5. Overlay operation using packed compare

```
for i = 1 to Sprite_Line_size
if A[i] = clear_color then
 Out_frame[i] = C[i] else
 Out_frame[i] = A[i]
```

where clear_color is the color surrounding the sprite in its 2D array in memory.

The output pixel is dependent on its color. The challenge is how to execute such data-dependent calculations on several pixels in parallel. For graphics simplicity in this example, we assume pixels are 16 bits, but the operation extends with double the parallelism for byte pixels.

MMX technology has a parallel compare instruction that generates a bit mask as its result. This instruction enables the data-dependent calculations to be executed on several data elements in parallel, and eliminates the need for any branch instructions. Figure 4 shows a bit mask result of the MMX parallel compare instruction. The result of the instruction has all 1s in elements where the relation tested for is true and all 0s where the relation tested for is false.

The mask is used as the source in logical operations to select desired data from the packed elements. In Phase 1 of Figure 5, the mask is a result of comparing four pixels from the sprite array with the clear color. Wherever the pixel from the sprite array belongs to the sprite, that is, is not equal to the clear color, the result of the comparison is 0. Wherever the pixel from the sprite array equals the clear color, the result is all 1s. This mask consists of 0s corresponding to the pixels from the actual sprite and 1s corresponding to pixels of the clear color. In Phase 2, this mask is used as an operand of the logical operation (A and complement of mask), which selects from the sprite array the real sprite pixels and zeros-out the surrounding pixels. The logical operation (C and Mask) selects from the scene the pixels that will not be cov-

ered by the sprite and zeros-out the pixels that will be covered by the sprite. The combination of these two results with a logical OR operation performs the overlay.

Changing Data Types

MMX TECHNOLOGY USES TWO SETS OF instructions—Pack and Unpack—to convert between the MMX data types. Unpack instructions take small data types and produce large ones (e.g., converting 16-bit to 32-bit words). Unpack instructions take two operands and interleave them. A user who simply wants to Unpack 16-bit into 32-bit words can take one operand of 16-bit information and interleave it with another operand filled with 0s. The result is 32-bit words with 0 in the most significant bits. In Figure 6, elements a0 and a1 from the packed word vector (A) are interleaved with a 0 vector. The result is a0 and a1, each

Operation	Latency	Throughput
Arithmetic logic unit	1	1
Multiplier	3	1
Shift/pack/unpack	1	1
Memory access	1	1

Table 2. Cycle count of the MMX instructions in the Pentium processor

addresses. On the other hand, this organization doesn't work for operating on columns. To be able to execute the same operation on all elements of a single column in parallel, the array needs to be transposed so columns become rows.

The Unpack instruction can be used to transpose an array ($B = A^T$) so the columns are converted into rows (see Figure 7). Transposition with MMX technology is a two-step process, and we assume in this example the elements in
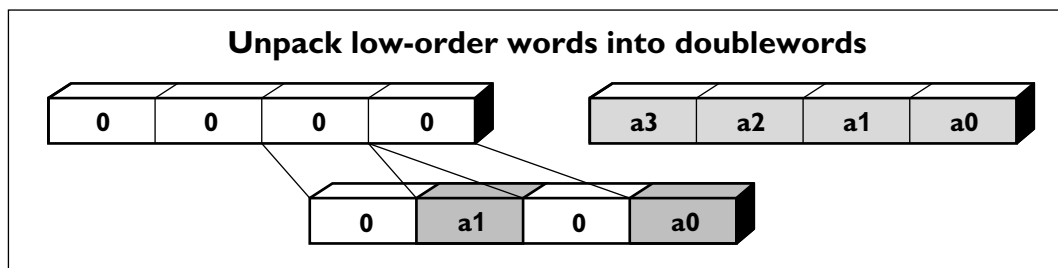

Figure 6. MMX technology Unpack high and low instructions on word data type

occupying a 32-bit element, that is, a0 and a1 were unpacked from a word data type to a doubleword data type. Unpacking is often used when 8-bit data pixels loaded from memory need to be expanded to 16-bit precision for the intermediate computations.

The Pack instructions are useful for converting data from a larger data type to the next smaller data type. Following the last example, if 8-bit bytes were expanded to 16-bit for intermediate calculations, the final results could be packed back into 8-bit bytes before the results are stored back to memory.

The Unpack instructions are also very powerful when data organized in one format in memory needs to be rearranged during an algorithm's course to expose parallelism amenable to MMX technology. An example is the Inverse Discrete Cosine Transform (IDCT) [6] used in, for example, the JPEG color-image decompression algorithm. The JPEG algorithm takes a 2D array of data and operates first on rows of data, then on columns of data. An array is usually laid out in memory one row after another. MMX technology is ideal for manipulating rows in this type of organization, as row elements reside in subsequent

the array are word (16-bit) values. In Phase 1, the Unpack instruction is used to interleave the word elements of adjacent rows; in Phase 2, the results of the first phase are unpacked again, this time using doubleword (32-bit) Unpack instructions to create the desired outputs.

MMX Integration into the Intel Architecture

MMX technology is fully compatible with the existing IA, meaning no new mode or state was created and all existing PC designs and operating systems can work with a new processor with MMX technology. From the instruction perspective, compatibility was easy because MMX instructions are defined to be integer instructions. From the data perspective, compatibility was a challenge because MMX data types are 64-bit packed integers, and there are no such integer registers on existing IA processors. The solution was achieved by mapping the MMX data types to the existing floating-point registers, which are 80 bits wide (see Figure 8). When MMX data is needed, the processor uses the floating-point registers as 64-bit-only packed integer registers. This mapping is all done internally in the processor, while the external world (operating system and PC) sees only the
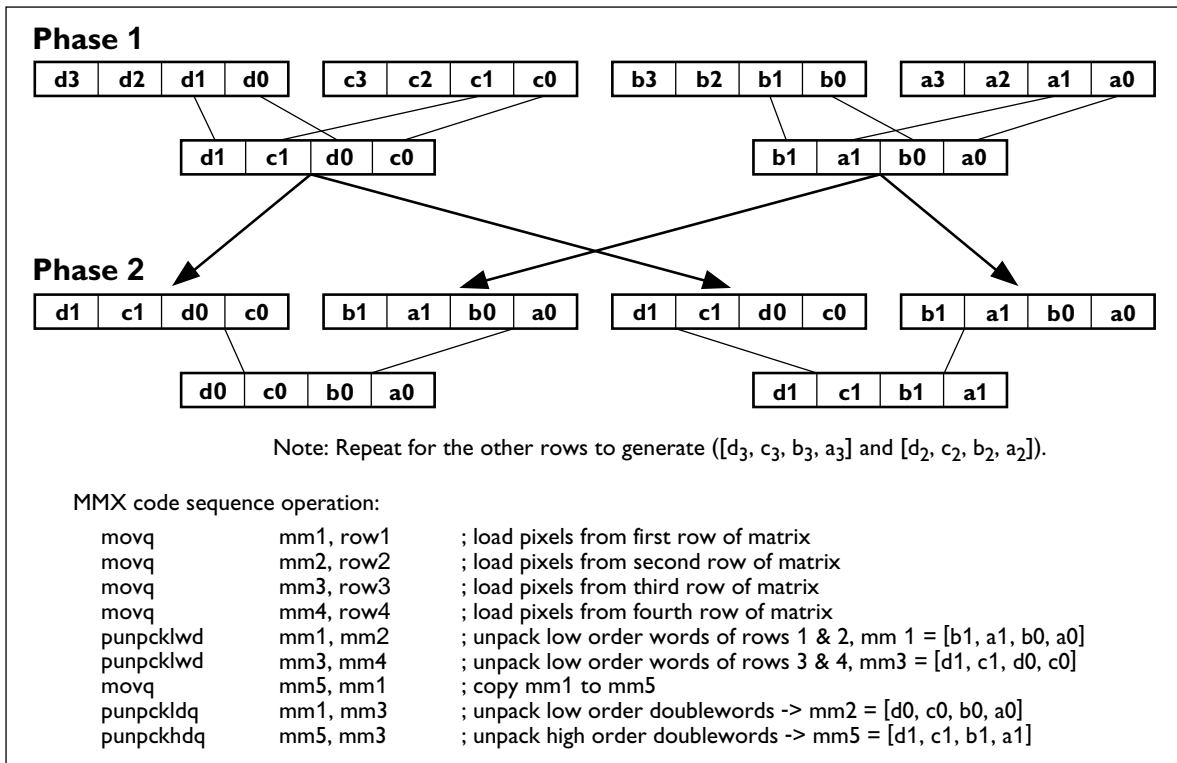
Phase 1

| d3 | d2 | d1 | d0 | | c3 | c2 | c1 | c0 | | b3 | b2 | b1 | b0 | | a3 | a2 | a1 | a0 |

| d1 | c1 | d0 | c0 | | b1 | a1 | b0 | a0 |

Phase 2

| d1 | c1 | d0 | c0 | | b1 | a1 | b0 | a0 | | d1 | c1 | d0 | c0 | | b1 | a1 | b0 | a0 |

| d0 | c0 | b0 | a0 | | d1 | c1 | b1 | a1 |

Note: Repeat for the other rows to generate ($[d_3, c_3, b_3, a_3]$ and $[d_2, c_2, b_2, a_2]$).

MMX code sequence operation:

```
movq        mm1, row1    ; load pixels from first row of matrix
movq        mm2, row2    ; load pixels from second row of matrix
movq        mm3, row3    ; load pixels from third row of matrix
movq        mm4, row4    ; load pixels from fourth row of matrix
punpcklwd   mm1, mm2     ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd   mm3, mm4     ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq        mm5, mm1     ; copy mm1 to mm5
punpckldq   mm1, mm3     ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq   mm5, mm3     ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```

Figure 7. Matrix transposition using Unpack instructions

physical floating-point registers. If the operating system needs to switch tasks suddenly, it saves the floating-point state (which may hold MMX data) and starts the new task. When the interruption is finished, the operating system switches back, restores the floating-point state (which may hold MMX data), then resumes where it left off. No physical new registers, condition codes, or events are added to support MMX technology. Having no additional new states allows all existing operating systems, including Windows, OS/2, and Unix, to work with MMX technology without being aware the technology is in the processor.

MMX TECHNOLOGY DEFINES EIGHT 64-BIT general-purpose registers laid over the floating-point registers. Each register can be directly addressed within the assembly by designating the register names MM0–MM7 in MMX instructions. These registers are used only for holding MMX data. Remember that MMX instructions are integer instructions operating on packed fixed-point integer data loaded into the floating-point registers. This scheme means all the registers on an IA processor (8-integer registers, 8-floating-point registers) can be put to use with MMX technology, getting the greatest benefit from the available registers on the processor.

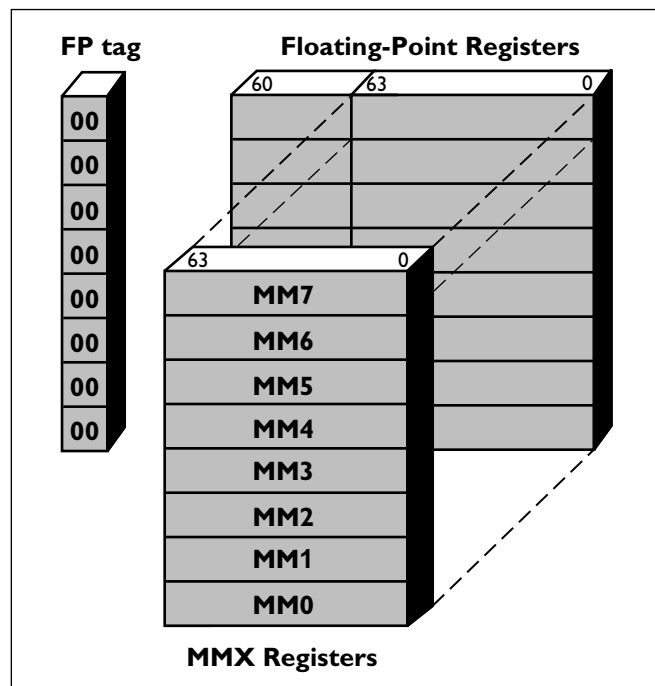MMX data values are put in the low-order 64 bits (the

FP tag **Floating-Point Registers**

60 63 0

00
00
00
00

63 0

00 MM7
00 MM6
00 MM5
00 MM4
MM3
MM2
MM1
MM0

MMX Registers

Figure 8. Mapping MMX registers to the floating point registers

Figure 9. MMX pipeline structure in the Pentium processor

packed integers, so it makes sense to use the computing hardware in the integer pipelines. The floating-point registers are used simply to hold the 64-bit packed-data types. MMX instructions—with the exception of the `multiply` instructions—execute in one cycle. (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 software loop unrolling, a throughput of one MMX `multiply` per cycle is achievable. Table 2 summarizes the latency and throughput of MMX instructions in the Pentium processor with MMX technology.

THE PENTIUM PROCESSOR CAN ISSUE TWO INTEGER instructions every clock cycle, one in each of the integer pipelines. During execution of any given instruction, the next two instructions are checked and, if possible, are issued so the first one executes in the first pipe and the second in the second pipe. If it is not possible to issue two instructions (e.g., the second instruction needs the result of the first instruction), the first instruction is issued to the first pipe and no instruction is issued to the second pipe. The second instruction waits for the next cycle and becomes the first instruction in the next possible pair. In a Pentium processor with MMX technology, a pair of instructions that can be executed in parallel can be two-integer instructions (as on the regular Pentium processor), one-integer instruction, and one MMX instruction, or two MMX instructions.

The basic integer pipeline structure of such a Pentium processor comprises the following stages:

- Instruction prefetch (PF)
- Instruction fetch (IF)
- Instruction decode (D1)
- Instructions paring and dispatch (D2)
- Execution and memory access (E)
- Writeback (WB)

Instructions with execution latencies of more than one cycle stay in the execute stage till they finish. For MMX instructions, the pipeline structure is slightly different, following this order:

mantissa) of the 80-bit floating-point registers. The exponent field of the corresponding floating-point register (bits 64–78) and the sign bit (bit 79) are set to 1s, making the value in the register a Not a Number (NaN) or infinity when viewed as a floating-point value. This helps reduce confusion by ensuring an MMX data value will not look like a valid floating-point value. MMX instructions access only the low-order 64 bits of the floating-point registers and are not affected by the fact they operate on invalid floating-point values.

Dual use of the floating-point registers does not preclude applications from using both MMX and floating-point code. Inside the application, the MMX code and floating-point code should be encapsulated in separate routines by the programmer. After one routine completes, the floating-point state is reset and the next routine starts. However, it is generally not a good idea to use the registers for floating-point and MMX data at the same time because values in floating-point registers are interpreted differently when accessed by floating-point instructions or by MMX instructions.

Pentium Processors with MMX Technology

Intel calls the first implementation of MMX technology on a Pentium processor the "Pentium Processor with MMX Technology." Here, we look at the Pentium processor because MMX technology is built into the basic processor design. The Pentium processor is an advanced superscalar processor (meaning it can handle two or more instructions simultaneously). Built around two general-purpose integer pipelines and a pipelined floating-point unit [3], it can simultaneously execute two integer instructions or one floating-point instruction. A software-transparent dynamic branch-prediction mechanism minimizes pipeline stalls due to branch instructions.

MMX instructions were architected to run in the integer pipelines despite the use of the floating-point registers to hold data. Keep in mind that MMX instructions operate on

Our goals were to exceed performance of Intel Architecture code by 100% to 300% on kernels (the basic loops of multimedia applications) and to provide an overall performance boost of 50% to 100% on multimedia applications.

- Instruction prefetch (PF)
- Instruction fetch (IF)
- Instruction decode (D1)
- Instructions paring and dispatch (D2)

At this point in the pipeline, MMX instructions continue down a different path:

- MMX operands read and write (MRW)
- MMX execution (Mex)
- MMX writeback (WB)

Additional stages are provided for the pipelined `multiply` instructions, which have a longer latency. This different pipeline path for MMX instructions is illustrated outside the box in Figure 9.

With the new MMX instructions, the instruction decode logic had to be modified to decode, schedule, and issue the new instructions at a rate of up to two instructions per clock cycle. The MMX opcodes map to the IA 0Fh (hexadecimal representation) prefix extension table, which is rarely used in existing software. Therefore, decoding these instructions in the original Pentium processor design was slow, with throughput of two clock cycles per instruction. The instruction decoder was redesigned to quadruple the throughput of instructions with a 0F prefix, allowing two instructions per cycle throughput.

Multimedia and communications applications have fairly high data access rates. MMX technology allows more computing to be done per clock cycle, so the processor needs to be able to move data more efficiently. Another way to express this efficiency is to say that because MMX technology increases computational bandwidth, we need to adapt memory bandwidth to maintain a balanced system. This memory bandwidth issue is addressed by the Pentium processor with MMX technology in two ways:

- The architecture uses 64-bit-wide packed integers that naturally take advantage of the 64-bit wide data bus on the Pentium processor.
- The code and data caches on the Pentium processor with MMX technology were doubled to 16KB each and operate at core frequency.

The memory bandwidth issue is also being dealt with by Intel at the system level with better chipsets (the support chips surrounding the CPU in a PC) and new peripheral buses for disks, CDs, and graphics cards, such as the Peripheral Components Interconnect (PCI) [8] and the Accelerated Graphics Port (AGP) [12].

MMX Read/Write Stage

ONE OF THE IMPORTANT MODIFICATIONS TO THE pipeline of the Pentium processor with MMX technology was incorporation of a special MMX operands read/write stage, in which source operands residing in MMX registers or memory are read. Also done in this stage is writing results from MMX registers to memory. As a result of this new stage, an instruction that performs a calculation and has one of its operands in memory (while the other comes from a register) takes only a single clock cycle to execute. With a regular non-MMX Pentium processor pipeline structure, such instructions as `ADD EAX, MEM` needed two cycles to execute. In the first cycle in the execute stage, the memory operand would be read from the cache (one-cycle latency assuming a cache hit). The actual calculation would happen in the second cycle. Thus, this instruction resides in the execute stage for two cycles, effectively stalling the pipe for one cycle. Breaking away the read operation for MMX instructions into a separate stage from the actual calculation operation avoids this stall. For example, the MMX instruction `PADDB MM2, MEM` would reside for one cycle in the MRW stage to read its memory operand (assuming a cache hit), then move on to the Mex stage to perform the calculation, freeing the MRW stage for the next MMX instruction.

MMX Technology Performance Boost

Our goals for the Pentium processor with MMX technology were to exceed performance of IA code by 100% to 300% on kernels (basic loops of multimedia applications) and to provide an overall performance boost of 50% to 100% on multimedia applications. For example the Intel Media Benchmark [9] shows a performance boost of over 65%.

The main challenge in evaluating the performance of a new architecture and instruction extension like MMX tech-
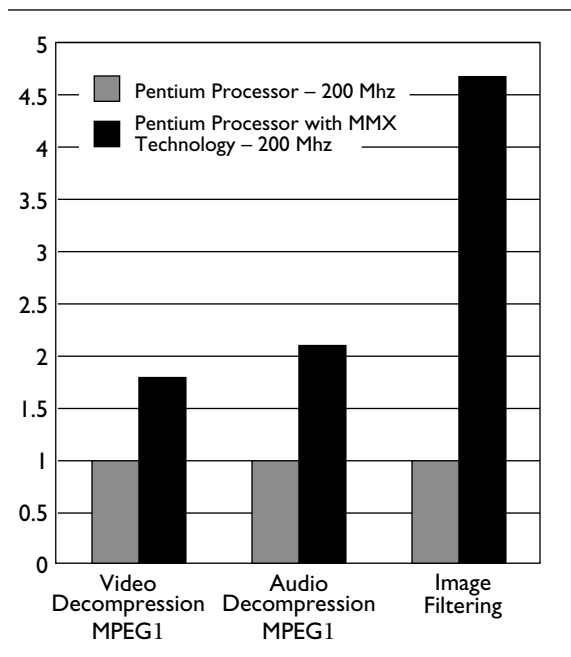
Figure 10. Intel media benchmark speedup with MMX technology

nology on any implementation is generating reasonable workloads for performance analysis and establishing a performance baseline. (Only recently has the computer industry sought to generate a multimedia benchmark suite, and even if there had been one during MMX technology definition, it would not have had MMX code in it.)

S A RESULT, PERFORMANCE PROJECTIONS FOR the Pentium processor with MMX technology were at first based on the kernel level. A set of kernels corresponding to several multimedia domains were coded using the new MMX instructions. Kernels included general-purpose algorithms, like the Fast Fourier Transform (FFT), the Finite Impulse Response (FIR) filter, and a vector-dot product, which are all basic signal-processing primitives, and domain-specific algorithms, like IDCT and Motion Compensation used in image and video decompression. The kernels were chosen to represent several key processing modules of multimedia applications: video, audio and speech compression, communications, 3D graphics, and image processing. The performance of these kernels was compared using a performance simulator to optimized versions of the same kernels—but without using MMX instructions.

However, MMX technology's most meaningful performance goals were on the application level. Various multimedia and communications applications were analyzed to understand where they spend most of their execution time. We found that most applications spend most of their execution time in a few basic kernels—in most cases in kernels that can be speeded up with MMX technology. We replaced these kernels with our MMX code versions and ran the application on our performance simulator and then on fully functioning silicon when it became available.

Using the internal performance counters feature in the Pentium processor and logic analyzers to probe the PCI and memory bus, the performance of the MMX implementation in several applications was analyzed in depth. As a result of this analysis, we learned that if parts of the applications were recoded to better utilize other aspects of the PC system, such as the cache memory hierarchy and the PCI bus, the performance advantages of MMX technology were further enhanced. The performance of applications with MMX technology as well as regular IA applications in some cases exceeded our expectations.

This approach to analyzing the performance of multimedia applications with MMX technology allowed us to achieve two goals:

- Refinement of the microarchitecture of CPUs implementing MMX technology
- Performance tuning of MMX software

Figure 10 shows the performance advantage of the Pentium processor with MMX on the Intel Media Benchmark [9], which includes a few multimedia applications that can be compared with and without MMX technology. The bars show the speed-up of the applications running on the Pentium processor with MMX technology over the same applications without MMX technology on the same processor. Some applications, like MPEG1 video decompression, speedup 80% with MMX technology, a relatively low speedup because this application spends substantial execution time in operations not speeded up by MMX technology, like accessing the CD-ROM drive or hard disk and writing data to the graphics memory. Improvements in these platform technologies and better graphics subsystems will eventually help PCs take full advantage of MMX technology. Other applications, like image filtering, speedup 370%. Image filtering spends most (more than 80%) of its time in the compute-intensive image-filtering kernel.

Early Work with Software Developers

Early on, we recognized that MMX technology success depends on the availability of exciting and compelling software using it. We therefore engaged industry software developers early in the MMX technology development program. Technical leaders in the software community worked with MMX technology in simulation environments to determine how they could best use the technology. Later,

We found that most applications spend most of their execution time in a few basic kernels—in most cases in kernels that can be speeded up with MMX technology.

when sample processors became available, we engaged a broader range of software developers. At first, we focused on developers supplying the tools and building blocks for multimedia and communications applications, such as 3D engines and audio coders. Then we engaged the larger software developer community.

New processors with MMX technology will be shipping into a user base that is mostly traditional IA (often called Intel 386 architecture). Since this base will have processors with and without MMX technology, we wanted to make sure developers could create applications running on both. To enable applications with MMX technology code to also run on processors without MMX technology, we added a feature bit to detect the existence of MMX technology. This detection operation is done by executing the `CPUID` instruction and checking a set bit in the result, giving software developers the flexibility to determine which software should be run. During run time, the software can query the microprocessor to determine if MMX technology is supported and execute MMX code on processors with the technology. Thus, only one version of the software needs to be sold, but including two versions of the main compute-intensive kernels: a regular IA version and an MMX technology version. Binary code size needs to grow to support this duality. Since MMX code is mainly applied to small, tight compute-intensive kernels, most applications we have studied need small total code growth to support both MMX and non-MMX technology versions for the kernels. Growth of less than 10% for the whole binary is common.

MMX Technology, Connected PCs, and the Internet

The Internet has brought the PC user visual impact and interactivity previously available only on a so-called multimedia PC. With a standard Web browser, users can download pictures and sound, seeing them locally on their PCs. In addition, users can manipulate video and 3D worlds, interacting with them both locally and on other PCs across the Internet. MMX technology provides new enhanced performance for some of these connected applications.

Establishing a voice phone call over the Internet (sometimes called Web telephony) requires the user's voice to be digitized and then compressed into as few bits as possible.

The compressed voice data can then be transmitted over the Internet without needing a lot of bandwidth. On the receive end of the call, the voice data must be decompressed to reproduce the voice signal. Voice-stream compression and decompression uses filtering and transform techniques, both of which are `multiply`-intensive. MMX technology can perform four `multiplies` in parallel, providing a significant performance boost for audio compression. Using a Pentium processor with MMX technology, a software application can implement the phone function (including compress/decompress activities) using only 20% of the cycles of the main processor.

VIDEO CONFERENCING HAS BEEN ADAPTED TO standard phone lines and Internet standards and is a step up in complexity over the voice call because video images are sent with voice information. The key to video is good compression, because available bandwidth is limited. The compression method—estimating the change from frame to frame, rather than transmitting every picture frame—can be speeded up by using MMX technology for computing absolute differences, as described earlier.

3D animation is becoming a useful way to help users "see" complex objects and information. Virtual reality modeling language (VRML) engines are designed to work with a minimal set of information so data can be transmitted over the Internet using as few bits as possible. The engine must take the small data set and render the 3D object. The saturating arithmetic in MMX technology can, for example, speedup Gouraud shading, a standard technique for making images look realistic.

Some picture formats used in the Internet also benefit from MMX technology. JPEG is a format that compresses and decompresses pictures with a technique similar to the IDCT, a technique that is multiply-intensive and requires arrays to be transposed for 2D processing. MMX technology can perform four `multiplies` at once and can use its `Unpack` instruction to quickly transpose data arrays. MMX technology helps speedup JPEG decompression engines used for JPEG files downloaded from the Internet.

The Java language is quickly becoming a standard programming language for applications executing over the

Internet. Portability is one of the language's main concepts, so downloaded code runs on any platform regardless of the CPU. This portability is achieved by executing an interpreter program on the receiving platform to execute the Java code. The problem with this solution is that interpretation is a slow process and as a result Java programs have low performance. A few techniques have been suggested to overcome this problem. One is use of just-in-time compilers that compile the program into the assembly code language understood by the CPU on the receiving platform instead of interpreting the program. Although the resulting code runs much faster, the compilation generates a delay until the program can start executing, worsening the latency problem already characterizing the Internet. Another solution is defining a set of media classes for the Java language. Such classes have been announced and are currently under development. They would be resident on the PC, be written in the assembly code language understood by the CPU on the receiving platform, and be used by the Java application at run time for performing graphics and animation. These classes can benefit from MMX technology and, as a result, execution of Java applications would be speeded up.
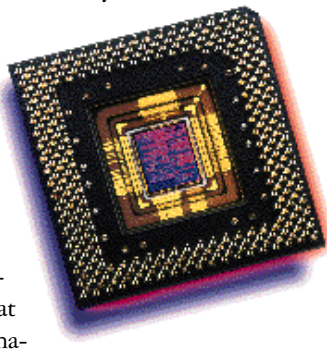
Conclusions

MMX technology implements a new high-performance architectural technique that enhances the performance of IA microprocessors for multimedia and communications applications. These applications' algorithms are compute-intensive, process a lot of data, use small data types, and provide lots of opportunities for parallelism. MMX technology brings more power to these algorithms by adding data types and instructions that can process data in parallel. This parallel processing is done while maintaining full compatibility with the existing installed base of operating systems and IA applications.

MMX technology is general by design and applicable to a variety of software media problems. Future media-related software technologies for use on the Internet should benefit from MMX technology, which will be integrated first into the Pentium and P6 processors, giving them an extra boost in capability. In the future, MMX will appear on all Intel processors.[1] (For more information, code examples, and applications, please see http://www.intel.com/drg/mmx.)

---

[1]No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this publication. Third-party brands and names are the property of their respective owners.

REFERENCES
1. Crawford, J., and Gelsinger, P. Programming the 80386. Sybex, San Francisco, 1987.
2. Diefendorff, K., and Allen, M. Organization of the Motorola 88110 superscalar RISC microprocessor, IEEE Micro 12, 2 (Apr. 1992), 40–63.
3. Intel Corp. Pentium Family User's Manual. Vol. 3, Architecture and Programming Manual. Intel Corporate Literature Sales, Mt. Prospect Ill., 1994
4. ITU-T. Recommendation H.263: Video Coding for Low Bitrate Communication. ITU, Geneva, United Nations, 1995.
5. Lee, R. Subword Parallelism with MAX-2. IEEE Micro 16, 4 (Aug. 1996), 51–59.
6. Loeffler, C., Lightenberg, and A., Moshytz, G. Practical Fast 1-D DCT algorithms with 11 multiplications. In Proceedings of the 1989 International Conference on Acoustics, Speech, and Signal Processing (Glasgow, Scotland). IEEE Computer Society Press, New York, 1989, pp. 988–991.
7. Peleg, A., and Weiser, U. MMX technology extension to the Intel Architecture. IEEE Micro 16, 4 (Aug. 1996), 42–50.
8. Shanley, T. PCI System Architecture. PC System Architecture Series, vol. 4. Mind Share, Richardson, Tex., 1993.
9. Slater, M. The land beyond benchmarks. Comput. Commun. OEM Mag. 4, 31 (Sept. 1996), 64–77. Intel Media Benchmark URL: http://pentium.intel.com/procs/perf/icomp/imb.htm.
10. Tremblay, M., O'Connor, J.M., Narayanan, V., and Liang, H. VIS speeds new media processing. IEEE Micro 16, 4 (Aug. 1996), 10–20.
11. Watt, A. 3D Computer Graphics. 2d. ed., Ch. 5.3, Addison-Wesley, Reading, Mass., 1993, pp. 131–136.
12. Yao, Y. AGP speeds 3D graphics. Microprocessors Rep. 10, 8 (June 17, 1996), 11–15.

ALEX PELEG (apeleg@iil.intel.com) is a senior computer architect in Intel's Israel Design Center Architecture Group.

SAM WILKIE (sam_wilkie@ccm.sc.intel.com) is program manager in Intel's Desktop Products Group.

URI WEISER (weiser@iil.intel.com) is director of Intel's Israel Computer Architecture Group.