Preface

Lin Chao
Editor
Intel Technology Journal

Welcome to the *Intel Technology Journal*. After a decade as an internal R&D journal, we're broadening our audience and decided to use the Internet for publication. This electronic, quarterly journal provides a forum for Intel's researchers and engineers to present their work directly to you. MMX technology is the most significant enhancement to the Intel instruction set since the introduction of the Intel386™ processor. This issue focuses on the research and technology developments underlying Intel's MMX technology.

The Story of Intel MMX™ Technology

By Albert Yu
Senior Vice President and General Manager, Microprocessor Products Group
Intel Corp.

Today, it's clear that Intel's MMX™ technology is a success. But the technology almost never happened. In the beginning, there was little management support for it. But a dedicated engineering team believed so strongly in the concept that they continued to drive the project. The project eventually spanned five years and four Intel sites. At its peak, more than 300 engineers worked to design, build, and test the technology. Despite geographical, linguistic, and time barriers, Intel employees from around the world worked together to create a great new product.

Planting the Seed

MMX technology was first proposed in 1992. For a year and a half, a small group of engineers tried to generate interest in their concept. Ultimately, a group of technical and marketing experts in the Microprocessor Products Group sensed the potential and supported the concept. I also saw its value and asked them to develop a detailed proposal.

A great deal of creativity and innovation went into the proposal, but acceptance within the rest of Intel was still slow. "Some people were ready to quit," remembers Uri Weiser, director of the Architecture group at the Israel Development Center (IDC) in Haifa and one of the people driving the project. Uri (named an Intel Fellow for work that included developing the architecture for MMX technology) co-authors a paper in this issue entitled "MMX™ Technology Architecture Overview."

Support Grows

The moment of truth came in February, 1994, when this team presented their findings and proposal at a Group Strategic Review. The presentation was attended by Chairman and CEO Andy Grove, President and COO Craig Barrett, and Chairman Emeritus Gordon Moore. For two hours, the merit of MMX technology was debated. Although the performance figures were impressive, there were questions about their validity. Some doubted that the architecture was viable. The deciding moment came when Andy asked a critical question about consumer benefits. The team was thrown off—nobody answered him immediately. Andy rejected the proposal, telling them to go back and do their homework.

The team followed Andy's advice. For the next two weeks they re-ran the tests, performed more simulations, and clearly spelled out the benefits of MMX technology. They presented the new results at a second Group Strategic Review three weeks later and the MMX technology project was approved.

Into the Pentium® processor

Since we had approval, we decided that MMX technology should go first into the Pentium processor and then into all future Intel processors. This was a huge risk for us.

Intel faced a brand-new technology where lots of details were not developed yet. Furthermore, it was going into the "crown jewel" of Intel's product line — the Pentium processor. Not since the Intel386™ processor had Intel made such significant enhancements to the instruction set. Intel's Israel Design Center in Haifa was chosen to design and build the MMX microprocessor. This too was a major risk for Intel. This was the first time Intel had developed a mainstream microprocessor outside the United States. But I was convinced that the team would pull it off successfully with the support of the rest of Intel.

Intel also took a significant risk in putting so much of its marketing prowess behind the new microprocessor. Our marketing group worked hard to persuade software vendors to create programs that took advantage of MMX technology. The result: at product introduction a large number of MMX technology-based software titles were available. The 57 new instructions increased the speed and quality of these multimedia applications such that they really shone.

Leaving a Legacy

MMX technology has been a resounding success in the marketplace. Many members of the MMX technology team have moved on to their next assignments, but their remarkable achievement will go down in Intel history as an example of how Intel's risk-taking values and constant innovation can pay off. This quarter's ITJ describes their research and development efforts in making MMX technology a reality.

.

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).

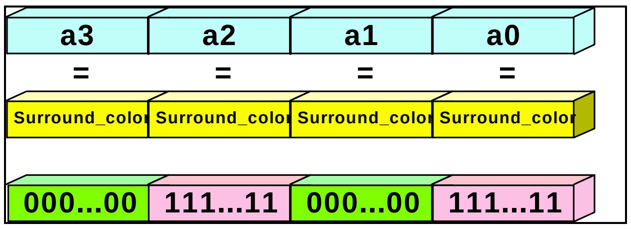

Figure 1. Packed Equal on Word Data Type

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.

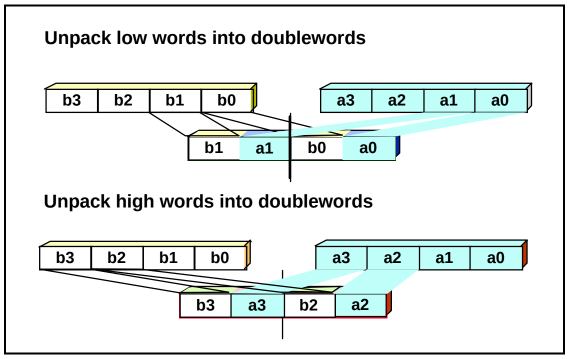
Unpack low words into doublewords
b3
b2
b1
b0
a3
a2
a1
a0
b1
a1
b0
a0
Unpack high words into doublewords
b3
b2
b1
b0
a3
a2
a1
a0
b3
a3
b2
a2

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

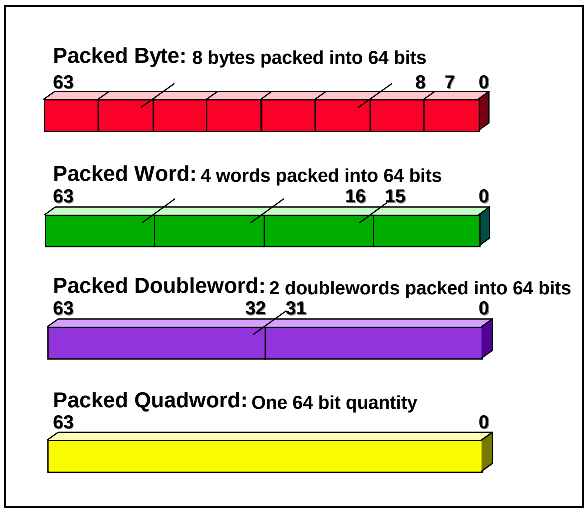
Packed Byte: 8 bytes packed into 64 bits
63
8 7 0
Packed Word: 4 words packed into 64 bits
63
16 15
0
Packed Doubleword: 2 doublewords packed into 64 bits
63
32 31
0
Packed Quadword: One 64 bit quantity
63
0

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).

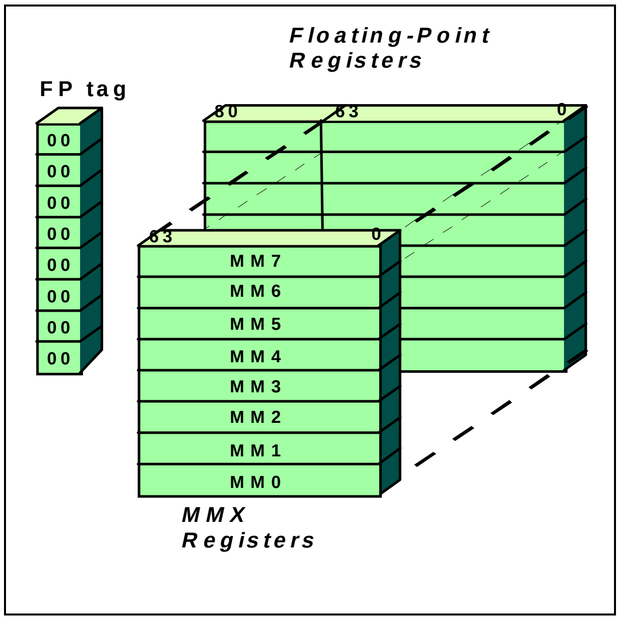

Figure 4. Mapping of MMX Registers to Floating-Point Registers

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.

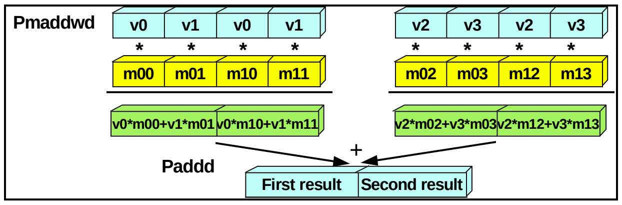

Figure 5. MMX Technology Matrix-Vector Multiplication

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.

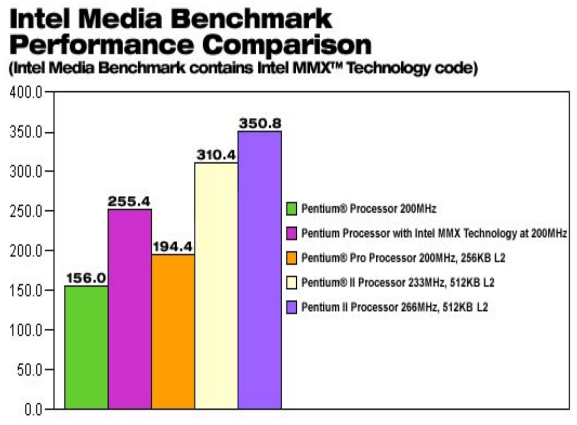

Figure 6. Intel Media Benchmark Performance Comparison

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.

MMX™ Microarchitecture of Pentium® Processors With MMX Technology and Pentium® II Microprocessors

Michael Kagan, IDC, Intel Corp.
Simcha Gochman, IDC, Intel Corp.
Doron Orenstien, IDC, Intel Corp.
Derrick Lin, MD6, Intel Corp.

Index Words: MMX™ technology, multimedia applications, IA extensions, Pentium® processor

Abstract

The MMX™ technology is an extension to the Intel Architecture (IA) aimed at boosting the performance of multimedia applications. This technology is the most significant IA extension since the introduction of the Intel386™ microprocessor. The challenge in implementing this technology came from retrofitting the new functionality into existing Pentium® and Pentium® Pro processor designs.

The main challenge was how to incorporate the new instructions while also keeping upcoming products on the Intel performance curve. Both projects had to deliver higher performance than their predecessors on legacy applications, using both frequency gain and CPI (Clocks Per Instruction) microarchitecture improvements.

On the other hand, new instructions had to be implemented in a cost-effective way, e.g., provide a breakthrough performance boost on multimedia applications while maintaining reasonably low die size cost. Moreover, the Pentium processor with MMX technology and Pentium® II processor, being the first microprocessors to implement the new Instruction Set Architecture (ISA), had to deliver superior multimedia performance to demonstrate that the benefit of the ISA extension would be compelling enough for Independent Software Vendors (ISVs) to develop software using these new instructions and fuel up the software spiral.

The new instructions operate on packed data types (single operand represents more than one datum) and use a flat register file that is architecturally aliased to an existing register file of the Floating-Point (FP) stack. This definition allows a variety of implementation alternatives.

Additional changes were introduced in the micro-architecture of the predecessor microprocessor in order to stay on the performance curve, improving the frequency and clock per instruction performance.

Introduction

During the ramp of the Pentium® processor in 1993, it became evident that the home market was becoming a major consumer of PCs, with a major boost coming from multimedia applications.

Traditionally, multimedia applications were supported by expansion hardware with dedicated software, thereby increasing the cost of the machine and lacking common standards. Engineers in the Intel Architecture group envisioned the need of executing operations for multimedia on the core CPU. This would establish a standard for the industry, reduce the cost of the system, and free up motherboard expansion slots.

A distinct characteristic of multimedia applications is the execution of the same operation on multiple small-size data items (e.g., 8 and 16 bits). The Single Instruction Multiple Data (SIMD) architecture provides a cost-effective solution for such applications, and therefore it was decided to extend the IA with 57 new MMX™ SIMD-type instructions.

At the time of this decision, two design projects were in their initial development stages: a high-end Pentium processor, and the Pentium® II processor, a Pentium® Pro processor compaction, both based on Intel's 0.35u CMOS process. In order to allow a fast ramp and a top-to-bottom penetration of the new extensions into the PC market, it was decided to incorporate new instructions in both projects and have them become the flagships of the new architecture extension.

At that time, the Pentium and Pentium Pro processors were both in advanced development stages with a much more mature database and silicon experience. In order to stay on the performance curve and catch up on frequency, we had to set a more aggressive frequency goal than our predecessors and also improve CPI performance. In the Pentium processor with MMX technology, this resulted in restructuring the entire machine by adding one more stage to the processor main pipeline. The Pentium II processor design team improved the performance of graphics applications and achieved a higher frequency through less aggressive architectural changes.

Both design teams delivered excellent results. The Pentium processor with MMX technology achieved both its CPI and frequency goals. It is 20% higher in frequency (running at 233MHz in production) and 15% faster on CPI than other Pentium processors. The Pentium II processor significantly improved the performance of graphics code and achieved a 300MHz frequency at introduction. The speedup goal for multimedia applications was achieved as well. Most applications using the new instructions improved by a factor of 1.6X, with some having improved up to 4X.

Pentium Processor With MMX Technology Microarchitecture

In order to exceed the performance of its predecessor, the design team had to improve both the frequency and CPI performance of the microprocessor. Both of these goals could be achieved with microarchitecture changes implemented in the new processor.

Frequency Speedup

Frequency is the most significant factor that determines the performance of a microprocessor and is a major (and sometimes only) performance indicator used by customers. Therefore, it was not possible to come up with a new product running at a lower frequency than its predecessor.

The frequency improvement of a product approaches asymptotically the architectural limit of the device by cleaning up escapes and by making slight design improvements in critical paths. Therefore, in order to match a predecessor's frequency, a product that comes to market later must have higher architectural frequency limits. Figure 1 illustrates frequency improvement trends for the Pentium processor and Pentium processor with MMX technology.

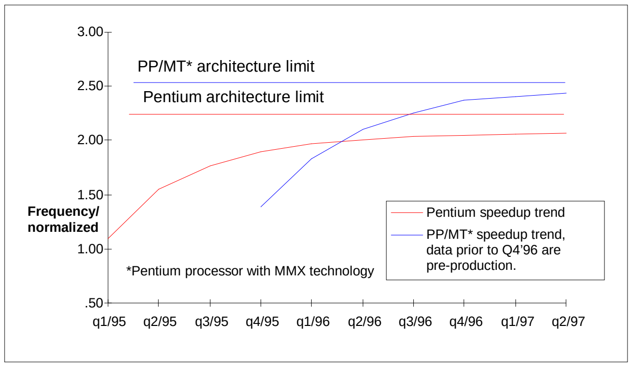

Figure 1. Frequency Improvement Trends

In order to improve the architectural limit of the device, we had to identify and resolve the major speed bottlenecks of the Pentium processor's architecture. After a thorough analysis, two major bottlenecks were identified: the decoder and the data cache access. The two bottlenecks were dependent. In other words, resolving one of them would help to speed up the other one. We decided to resolve the decoder bottleneck, since it was simpler and less risky, and it would also allow a smooth implementation of MMX instruction decoding. The Pentium processor execution pipeline originally consisted of five pipeline stages: Pre-fetch (PF), Decode1 (D1), Decode2 (D2), Execute (E), and Writeback (WB). We added an additional pipeline stage in the front end of the machine, rebalanced the entire pipeline to take advantage of the extra clock cycle, and added a queue between the F and D1 stages to decouple freezes, which are the most critical signals generated in every pipeline stage. Figure 2 illustrates the difference between the original Pentium processor pipeline and the MMX technology pipeline.

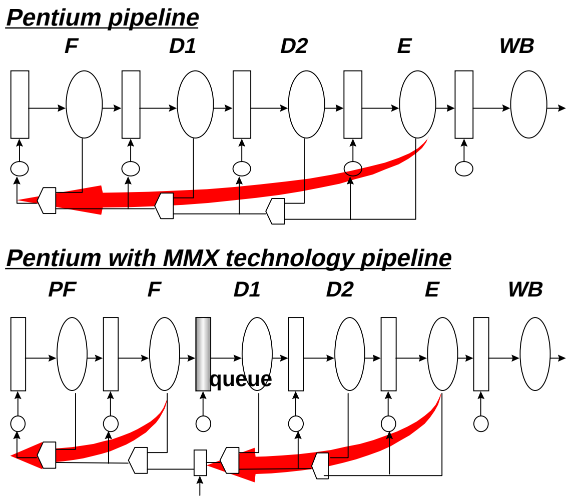

Figure 2. Pentium Processor and Pentium Processor With MMX Technology Pipeline

An additional clock cycle in the front end of the pipeline resolved the decoder speed bottleneck and reduced fanout for the data cache freeze (generated in the E stage), which in turn relaxed a requirement for this freeze signal. This was the first step in the resolution of the data cache bottleneck.

The next step was to improve the timing of the data freeze signal generated by the data cache. The cache access path starts with address generation in the D2 stage, followed by a subsequent cache access in the E stage. The entire path was redesigned to self-time pipelined execution with time borrowing between the stages. The address generation logic was changed, incorporating simplified and faster adders, thereby allowing faster address generation.

The third step was the cache circuit architecture. It was performance-crucial to execute a single clock read and write operation in each cache port. As a result, the Pentium processor's cache access windows were designed to support two access windows per clock, as illustrated in Figure 3.

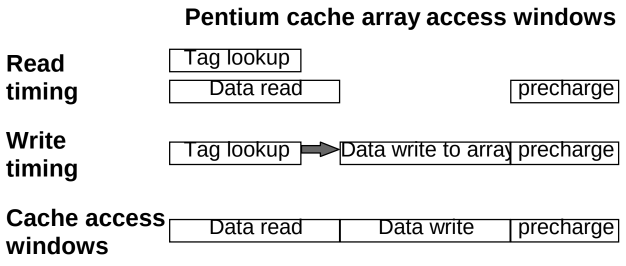

Figure 3. Pentium Processor's Cache Array Access Windows

Although read and write operations to the same port were never performed in the same cycle, cache timers had to support two access windows, thereby limiting the overall cache access time. On the other hand, since read and write operations never happen in the same clock to the same port, both access windows could never be active in the same clock cycle. In other words, during a read operation, no data access could be performed in a write access window and vice versa. Therefore, we decided to have just one data access window in the front end of the cycle (e.g., read window timing) and use it for both read and write accesses. The read access works as in other Pentium processors; it is a speculative operation and can be thrown away. Write access depends on the result of a tag lookup and cannot be executed if the same clock tags are looked up. Therefore, the Pentium processor with MMX technology implemented a cache store hit buffer. If a store hit is encountered at the cache lookup phase, the data is stored to this buffer. The actual store to the data array will be done at the data access window of the next write operation, while this window is idle. Meanwhile, before the next write, the data can be delivered from the store hit buffer to subsequent reads from this address. In other words, the Pentium processor with MMX technology pipelines write operations among each other. Each time a store is executed, the tag lookup is performed for the current store, while the data array is updated with data from the previous store. This way we could have only one data array access window, which allowed a significant speedup of cache access.

Figure 4 illustrates the Pentium processor with MMX technology's cache access windows architecture.

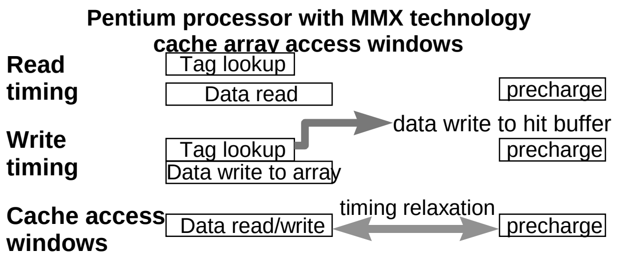

Figure 4. Pentium Processor With MMX Technology's Cache Array Access Windows Architecture

The solutions described above resolved major Pentium processor speed paths, allowing a frequency leap. Additional local changes were performed in every functional block to keep all the rest of the circuitry in line with this new goal.

In summary, the Pentium processor with MMX technology designers addressed two major bottlenecks at a global architecture level (adding a pipeline stage and re-balancing the entire machine), made few changes on the intermediate level (time borrowing between pipe stages for a specific operation), and implemented numerous local changes to keep the machine balanced. This top-down approach allowed us to achieve a 20% frequency boost over the original Pentium processor design.

CPI Performance

Although adding a pipeline stage improves frequency, it decreases CPI performance, i.e., the longer the pipeline, the more work done speculatively by the machine and therefore more work is being thrown away in the case of branch miss prediction. The additional pipeline stage costs decreased the CPI performance of the processor by 5-6%.

In order to stay on the performance curve, we had to gain back this loss and, in addition, speed up the machine further.

The Pentium processor with MMX technology's CPI performance was increased in three major ways:

1. Improved branch prediction. We implemented a more advanced branch prediction algorithm that was developed by the Pentium Pro processor design team. This algorithm improved the prediction of branches, which resulted in fewer miss-predictions of branches

and caused less work to be thrown away. On top of the Branch Target Buffer (BTB), we also implemented a Return Stack Buffer (RSB)—a dedicated branch prediction logic for call/return instructions. The combination of the updated BTB algorithm and the RSB improved CPI performance by about 8%. This helped close the performance gap opened while adding the new pipeline stage and gave us some advantage over the Pentium processor.

2. Improving core/bus protocols. The original Pentium processor design was tuned to a 1:1 ratio between the core and bus clocks. As a result, some price/performance tradeoffs that were made for a 1:1 clock ratio were not optimal for use when the gap between the core and bus frequency increased. Several enhancements were made by the design team to tune the protocols. Write buffers were combined into a single pool, thereby allowing both pipes to share the same hardware, the clock crossover mechanism was changed, and the DP protocol was completely redesigned to decouple core and bus frequencies. These improvements gained about a 5% CPI performance improvement and simplified the design and testing (e.g., crossover, DP protocols).

3. Creating larger caches and fully-associative Translation Lookaside Buffers (TLB). In general, increasing cache size is the most cost-effective way to improve performance. The Pentium processor with MMX technology increased the size of both caches from 8Kbyte to 16Kbyte and made them four-way set-associative. Fully-associative TLBs improved CPI to some extent, making address translation faster than in the original TLB design. Larger caches and fully-associative TLBs bought us about a 7-10% CPI performance improvement.

In summary, by improving the BTB, redesigning the core/bus protocol, and making larger caches, the Pentium processor with MMX technology achieved about a 15% higher CPI performance than the Pentium processor despite the CPI loss due to the additional pipeline stage.

MMX Technology Implementation

After setting the stage for frequency and CPI performance, we could incorporate the MMX instructions relatively straightforwardly.

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. The MMX opcodes are mapped to a 0F prefix, which is rarely used in previous IA native software. Therefore, decoding of these instructions in the original Pentium processor design was slow, with a throughput of two clock cycles per instruction. The Pentium processor with MMX technology decoder was redesigned to quadruple the throughput of 0F instructions, allowing two instructions per cycle throughput.

Additional modifications were made to the MMX technology pipeline to incorporate the MMX execute stage (MEX) and the MMX writeback stage. To improve the performance of MMX ARITH-MEM instructions, the integer-execute stage is used as an MMX "read-stage," where the source operands as well as the memory operands are read. As a result, an ARITH-MEM instruction is executed in a single clock cycle. Since the Pentium processor with MMX technology may pair an ARITH-MEM instruction with an ARITH instruction, it is equivalent to having three execution units (two ARITH, one LOAD) working in parallel, similar in concept to a Pentium II processor.

According to the MMX technology architecture definition, the MMX register file is aliased to the FP mantissa register file. It was decided to design dedicated hardware to execute the MMX instructions (the Munit). This unit has a dedicated MMX register file, capable of delivering four 64-bit operands and storing three 64-bit results in a single clock cycle. The Munit also incorporates the MMX execution units, which were defined and designed as a module, and which allowed the design to be shared with the Pentium II processor.

Clean partitioning of the MMX technology design and an additional pipeline stage in the decoder resulted in no speed issues associated with the new units. The area penalty for the Munit was small.

Pentium Processor With MMX Technology Block Diagram

The block diagram of the Pentium processor with MMX technology is shown in Figure 5, outlining parts that were redesigned for speed, CPI, and MMX technology.

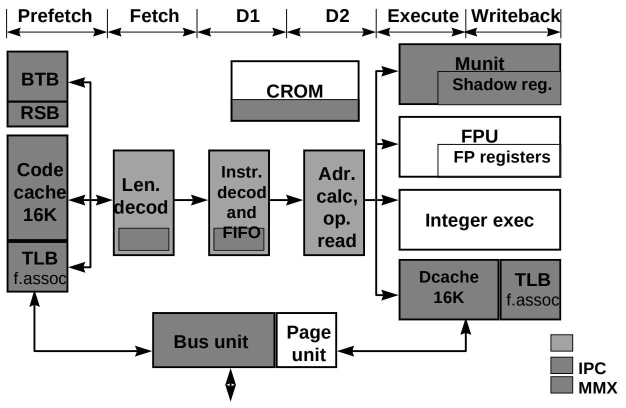

Figure 5. Block Diagram of the Pentium Processor With MMX Technology

Results

The Pentium processor with MMX technology design achieved its goals. The processor taped out in late 1995, and samples were delivered to customers less than a week after the first silicon. With six months of extensive silicon debug, we closed the frequency gap with the Pentium processor and, half a year later, achieved 233MHz in production, which is one bin above the Pentium processor's production frequency.

Figure 6 shows the actual speed improvement of the Pentium processor and the Pentium processor with MMX technology versus the anticipated trend.

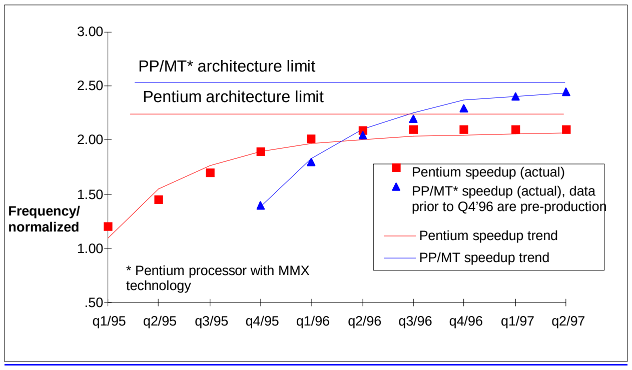

Figure 6. Actual Versus Anticipated Speed Improvement Trend

The Pentium processor with MMX technology also met its CPI goals. Figure 7 shows the CPI performance of the Pentium processor with MMX technology compared to the Pentium processor.

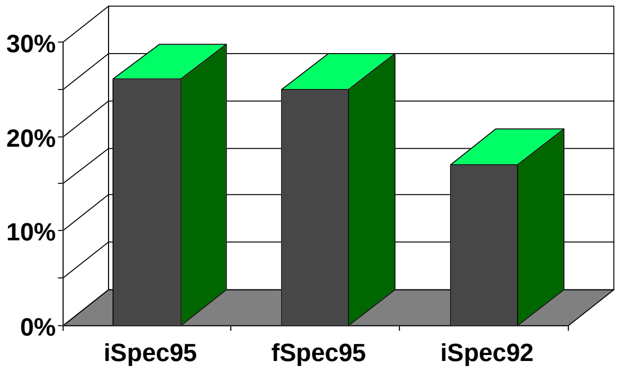

Figure 7. CPI Performance of the Pentium Processor with MMX Technology Compared to the Pentium Processor

And at last, multimedia applications gained significant performance using new instructions. Figure 8 illustrates the performance gain that can be achieved by several applications when using the new instructions.

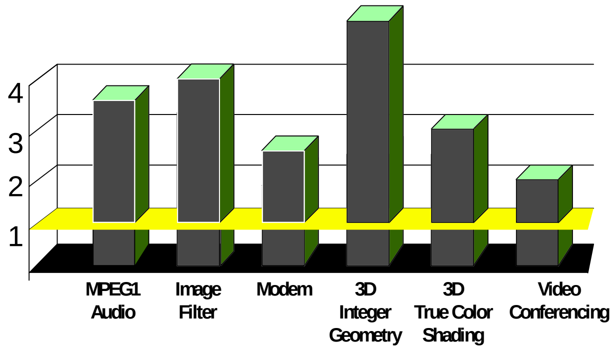

Figure 8. Performance Improvement Using New Instructions

Pentium II Processor Microarchitecture

While the Pentium processor with MMX technology made microarchitecture changes to improve frequency and performance as well as implement the MMX technology, the Pentium II processor improved upon the Pentium Pro processor's microarchitecture and brought MMX technology to a new level of performance. The Pentium II processor is based on the dynamic execution microarchitecture of the Pentium Pro processor. Changes were made in the Pentium II processor's microarchitecture to improve graphics performance and to implement MMX technology. In addition, the entire back-side bus interface that connects the processor to an off-chip second-level cache was redesigned to allow low-cost commodity SRAMs to be used as second-level cache. Doing so significantly reduced the system cost compared to the Pentium Pro processor's Multi-Chip Module (MCM) that houses the processor as well as the second-level cache. A higher frequency was achieved through aggressive circuit techniques and other changes.

Overview

The Pentium II processor is the second Intel microprocessor to implement MMX technology. The Pentium II processor's MMX technology implementation offers multimedia applications the benefits of an out-of-order execution, aggressive memory speculation, a superpipelined and superscalar microarchitecture, etc. These are the same features that the Pentium Pro microprocessor provides. The Pentium II processor supports two packed ALU operations, one packed shift, and one packed multiply operation. Pack and unpack operations are implemented by the packed shifter. The Pentium II processor allows packed shift and packed multiply to be executed concurrently.

MMX Technology Implementation

The Pentium II processor's microarchitecture is similar to that of the Pentium Pro microprocessor. The Pentium Pro processor's high-level microarchitecture consists of the following pipelines: instruction pre-fetch, length decode, instruction decode, rename/resource allocation, uop scheduling/dispatch, execution, writeback, and retirement. The length decoder was modified to include decoding for the new instructions. Some of the Pentium Pro processor's microarchitecture was modified to add MMX technology. The Instruction Decoder logic was modified to convert the new MMX instructions to Pentium Pro processor-specific uops (new Single Instruction Multiple Data [SIMD] uops were added to implement the new functionality). The renamer (RAT) was modified to handle MMX technology management of the floating-point stack because MMX registers are aliased onto the floating-point stack to avoid the need to modify operating systems. The resource allocator (ALLOC) was changed to provide static scheduling binding for the new SIMD uops. A new execution unit, the SIMD unit (MMX instructions are primarily SIMD instructions), was added. The Reservation Station (RS) was changed to accommodate the new SIMD 64-bit datapath. In addition, minor changes were made to expand some buses to accommodate the 64-bit MMX technology requirement.

Performance data showed a need for a dual execution pipeline for MMX instructions. Of the five execution ports (port 0 - FP, integer; port 1 - integer; port 2 - load; port 3 - store address; and port 4 - store data), it was decided to put the SIMD unit in ports 0 and 1. Due to area constraints, the SIMD unit implemented only one shifter and one multiplier. The multiplier and shifter are on different ports so that the two operations can execute in parallel. Arithmetic and logical execution hardware was duplicated to provide concurrent execution. The three-clock latency, fully pipelined multiplier was placed in port 0 because the Reservation Station (RS) already had logic to handle a multiple clock latency in port 0.

Floating-point registers are stacked. Although multimedia registers are aliased onto the floating-point stack, they are not used as stack registers. Like integer registers, multimedia registers are general purpose, randomly-accessed registers. The RAT treats the stack-based floating-point registers and the randomly-accessed multimedia registers the same way in terms of register renaming. This simplifies the floating-point rename logic. However, this approach only works if the floating-point Top-of-Stack (TOS) is zero. All floating-point registers are converted from stack-based logical registers to real logical registers by adding the stack register number with the floating-point TOS. If the floating-point TOS is zero, the stack adjustment is transparent to multimedia registers. For all floating-point and multimedia registers, the RAT converts the stack-based register number into a logical register number by factoring in the TOS reference. The logical register number is used to read-write the floating-point rename tables. In addition, each uop that presents stack-referenced registers as source/destination can modify the TOS. The RAT needs to perform the TOS reference and manipulation logic on the fly, as uops enter the rename pipeline stage.

Since the existing floating-point register logic works transparently for multimedia registers only if the floating-point TOS is zero prior to an MMX instruction, the rename logic will produce errors if the floating-point TOS is not zero. A microcode assist was created to correct the problem and redo the operation. An assist is a customer-invisible event that flushes out the machine and allows microcode to handle rare but difficult-to-handle problems. Since all MMX instructions zero the TOS, the assist needs to write the TOS to zero and restart the operation. Then, the operation sees zero TOS and the rename can occur correctly.

The MMX technology specification requires that all MMX instructions, except EMMS, set all floating-point stack registers to full. This logic is performed by the RAT as well. The setting of the stack valid bits to full is accomplished by setting all stack valid bits at once. This logic can get complicated when an MMX instruction is issued or retired at the same time as a floating-point instruction that performs a stack pop or otherwise clears one or more stack valid bits. To avoid the complexity, MMX instructions and floating-point instructions that check for stack valid or perform a pop are not allowed to issue or retire in the same clock. Due to timing constraints, the function of disallowing MMX instructions and floating-point instructions that check for stack valid or perform a pop to retire in the same clock is performed by another microcode assist. This assist can be avoided by following the MMX technology programming guidelines, i.e., placing an EMMS instruction between the last MMX instruction and the first floating-point instruction.

Challenges were encountered in the Instruction Decoder (ID) when adding functionality to decode the MMX instructions. All MMX instructions are placed on the 0F opcode map. However, these instructions are not arranged in blocks or in regular patterns. Opcode holes are sprinkled among the new instructions. The ID has three decoders: decoder 0, 1, and 2. Decoder 0 is a full-function decoder capable of decoding all instructions, with a maximum output of four uops. Decoders 1 and 2 are capable of decoding a subset of instructions that require only one uop. If the ID encounters an illegal instruction, a pair of uops are inserted into the uop stream by decoder 0, so the Re-order Buffer (ROB) knows to signal an illegal opcode fault. Thus, decoders 1 and 2 were not capable of handling illegal opcodes. There is a PLA (Programmable Logic Array) that determines whether an instruction can be decoded by any decoder. If not, the instruction must be decoded by decoder 0. This was a very speed-critical PLA and a new way of thinking was needed.

Since decoders 1 and 2 can decode one uop instruction, all register-register forms of MMX instructions, as well as the MOVD and MOVQ load instructions, can be decoded by these decoders. To reduce the minterm count of the critical PLA, we considered reducing the number of MMX instructions decodable by all decoders. This led to a noticeable performance loss. In order to avoid this performance loss while still meeting frequency goals, a new assist was created so that decoders 1 and 2 could handle illegal opcodes. This way, opcode holes and single uop instructions look the same to the two decoders. They all cause one uop to be generated. Illegal opcodes that are instruction holes in the MMX instruction opcode map are defined to generate a one uop assist call. This assist call instructs the ROB to flush the machine and causes an assist microcode flow to cause the processor to handle illegal opcode faults. All legal MMX instructions that decoders 1 and 2 can handle generate signal uops that implement the instructions. This optimization results in a minterm count increase of only 3 (out of 30). This enabled the PLA to meet the Pentium II processor's frequency target.

All MMX operations except load and store are executed by the SIMD unit. Each SIMD adder is capable of performing add, subtract, and compare of 8-byte, 4-word, and 2-doubleword data types. The adders are optimized to perform these operations with roughly the same speed as a normal 32-bit adder. The SIMD shifter can perform left and right parallel shifts of word, doubleword, and quadword. The SIMD multiplier performs the parallel multiply and multiply-add operations.

Summary

The challenges of implementing MMX technology in the Pentium II processor were in adding a major functionality while creating minimal disturbance to the base microarchitecture, which the processor inherited from the Pentium Pro processor. Furthermore, the changes made to the microarchitecture were such that the processor could run at the same frequency whether or not the functionality was present. To meet these objectives, novel microarchitecture, logic, and circuit techniques had to be used. The result of the Pentium II processor MMX technology implementation was a processor that can run at 300MHz at introduction, surpassing Pentium Pro processor performance.

Conclusion

The Pentium processor with MMX technology and Pentium II microprocessors are now available in the marketplace, introducing the next step in desktop, workstation, and server computing. The Pentium processor with MMX technology and Pentium II processors' acceptance into the marketplace is beyond expectations, and their ramp-up is the fastest in Intel history.

Both design projects successfully implemented the IA extension with a small cost in silicon area and with no architecture or logic bugs.

Acknowledgment

This paper presents the hard work of many talented people from the Pentium processor with MMX technology and Pentium II processor architecture teams. The persistence and dedication of these design and architecture teams made it possible to drive the definition and execution of these two outstanding design projects. Special thanks to design engineers from the Arizona design center who provided tremendous help to the Pentium processor with MMX technology design by sending about 40 engineers to work with us in Israel and taking some Pentium processor with MMX technology design tasks to execute in the Arizona design center.

Authors

Michael Kagan joined Intel in 1983 after his IDF military service. Michael graduated from the Technion in 1981 with a BS in Computer and Electrical Engineering. Michael worked in various areas of VLSI design, on all Intel microprocessor architectures, i.e., the 80890 (design engineer), 80387 (design engineer), 80860 (architecture and logic design manager), and Pentium processor with MMX technology (design manager). Michael's main professional focus is on processors and platform architecture and validation. His email address is michaelk@iil.intel.com.

Simcha Gochman received his BS and MS degrees in Electrical Engineering from the Technion in Haifa, Israel. From 1980 to 1983, he was an adjunct faculty member in the Department of Electrical Engineering at the Technion. From 1983 to 1984, he was with Refa'el, the Israeli Weapons Development Authority. Since 1984, he has been with the Intel Design Center in Haifa working on VLSI design and processor microarchitectures. His email address is simcha@iil.intel.com.

Derrick Lin is currently one of the senior logic designers on the Pentium II processor project. He received his BS

and MS degrees in Electrical Engineering from Stanford University and has been with Intel since 1991. His technical interests include all aspects of microprocessor and computer design, for example, microarchitecture, compiler, and high-speed circuit techniques. His email address is dlin@mipos2.intel.com.

Implementation of a High-Quality Dolby* Digital Decoder Using MMX™ Technology

James C. Abel, Intel Microcomputer Labs, Intel Corp.
Michael A. Julier, Intel Microcomputer Labs, Intel Corp.

Index Words: MMX™ technology, Dolby Digital Decoder, audio, multimedia.

Abstract

Dolby Digital is a high-quality audio compression format widely used in feature films and, more recently, on DVD[1]. PCs now offer DVD drives, and providing a Dolby Digital decoder in software allows decoding of Dolby Digital to become a baseline capability on the PC. Intel's MMX™ technology provides instructions that can significantly speed up the execution of the Dolby Digital decoder, freeing up the processor to perform other tasks such as video decoding and/or audio enhancement.*

A simple port of Dolby Digital to MMX technology using only a 16-bit data type introduces quantization noise that makes the decoder unsatisfactory for high-quality audio. However, MMX technology provides additional flexibility through 32-bit operations which, combined with other software techniques, allows the implementer to increase the audio quality of the decoder while still providing a significant speedup over implementations that do not use MMX technology. Intel has worked closely with Dolby Laboratories to define an implementation of Dolby Digital based on MMX technology that has achieved Dolby's certification of quality. This paper describes the research performed and the resultant techniques Intel used in creating its Dolby Digital decoder.

Introduction

Dolby* Digital is a transform-based audio coding algorithm designed to provide data-rate reduction for wide-band signals while maintaining the high quality of the original content [1]. MMX™ technology can be used to provide a processor-efficient implementation of Dolby Digital for a PC based on a Pentium® processor with MMX technology. It is important to maintain high audio quality, and Dolby Laboratories has developed a stringent test suite to ensure that a certified decoder indeed provides high quality. In addition, trained listeners evaluate prospective decoders using both test and program material. Only after a decoder has passed both the analytical and subjective tests is the decoder certified.

Intel's MMX instructions operate on 8, 16, and 32 bits. The human ear has an overall dynamic range of 120 dB and an instantaneous dynamic range of 85 dB [2]. The dynamic range of a binary value is 6.0206 dB per bit. Eight bits (48 dB of dynamic range, about that of AM radio) is obviously insufficient for high-quality audio. Sixteen bits (96 dB of dynamic range, as is used on Compact Disks) is usually considered high-quality audio, and we will accept this notion for this paper. However, due to rounding errors during the intermediate calculations, the accuracy at the output of a Dolby Digital decoder is significantly less than the accuracy of the intermediate values (assuming a uniform accuracy throughout the algorithm). This is typical in signal processing algorithms. Using 16 bits of accuracy uniformly through a Dolby Digital decoder is insufficient to pass the test suite. The challenge was to obtain both good execution speed and good audio quality. 32-bit floating-point numbers could be used throughout the data path and only use MMX technology for bit manipulation, but this would not be the most processor-efficient method. MMX technology provides integer operations that are more processor-efficient than existing floating-point operations; we strove to use the MMX instructions as much as possible.

The goal of this investigation was to find a minimal CPU implementation at an acceptable audio quality level. If the CPU requirements could be made small enough, Dolby Digital decoding entirely in software

[1] DVD is often referred to as Digital Versatile Disk or Digital Video Disk.

would be feasible, even in combination with other operations (such as video playback). In order to do this, we had to determine the accuracy required in the various stages of the Dolby Digital decoder while maintaining effective use of MMX technology. We found that by using the flexibility of the 16-bit and 32-bit data types in the MMX instruction set, we were able to increase the accuracy of the Dolby Digital decoder significantly beyond that of a simple 16-bit approach with only a small impact on CPU performance. We also found that MMX technology can be used to speed up the bit manipulation, dithering, and downmix sections of the decoder.

An additional benefit of performing the audio decode in software is the resultant flexibility possible in the audio subsystem. If the Dolby Digital decoder is in software, it is easier to route the decoded audio to other audio subsystems. For example, simultaneous mixing of the PC's system sounds (i.e., via the Microsoft Windows Wave Device API) with the decoded audio is possible.

Dolby Digital Decoder

A block diagram of the Dolby Digital decoder is shown in Figure 1 [3].

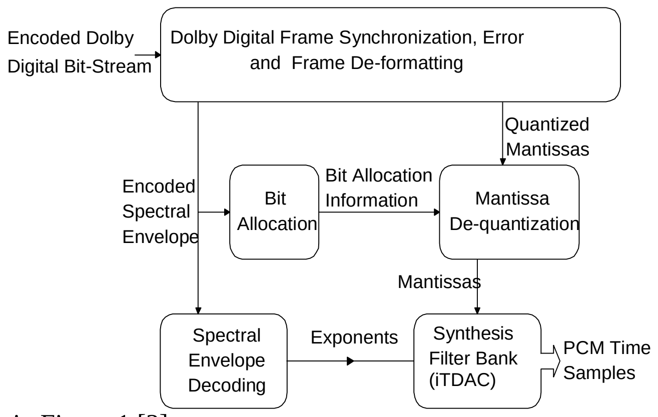
Encoded Dolby Digital Bit-Stream
Dolby Digital Frame Synchronization, Error and Frame De-formatting
Quantized Mantissas
Encoded Spectral Envelope
Bit Allocation
Bit Allocation Information
Mantissa De-quantization
Mantissas
Spectral Envelope Decoding
Exponents
Synthesis Filter Bank (iTDAC)
PCM Time Samples

Figure 1. The Dolby Digital Decoder

The Dolby Digital bit stream contains Synchronization Information (SI), Bit Stream Information (BSI), Audio Block information (AB), Auxiliary (AUX) information, and Cyclic Redundancy Check (CRC). See Figure 2 for the Dolby Digital bit stream.

During our investigation, each block was inspected to determine if it could benefit from MMX technology. The following operations benefit significantly from MMX technology:

- Bit Stream Parsing
- Scaling
- TDAC Transform (DCT twiddles, FFT, Windowed-Overlapped-Add)
- Dithering
- Downmixing

We will now describe the five major operations from the input to the output (Bit Stream Parsing, Coefficient Extraction, TDAC Transform, Dithering, and Downmixing). We will also describe how MMX technology was used to provide a speedup. General precision and performance enhancements will also be discussed.

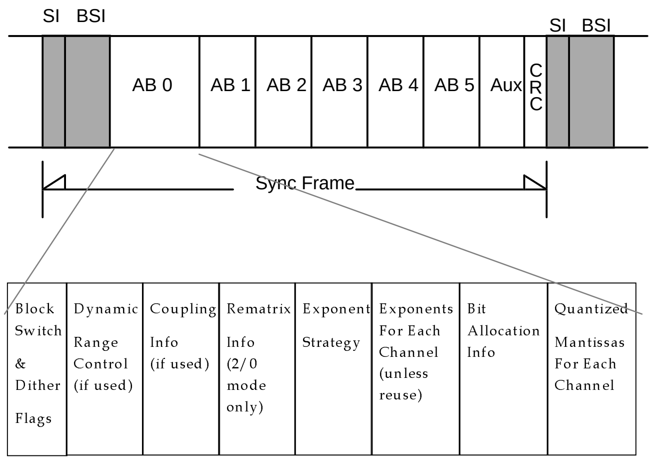
SI
BSI
AB 0
AB 1
AB 2
AB 3
AB 4
AB 5
Aux
CRC
SI
BSI
Sync Frame
Block Switch & Dither Flags
Dynamic Range Control (if used)
Coupling Info (if used)
Rematrix Info (2/0 mode only)
Exponent Strategy
Exponents For Each Channel (unless reuse)
Bit Allocation Info
Quantized Mantissas For Each Channel

Figure 2. Dolby Digital Frame and Audio Block

Bit Stream Parsing

Each audio block (AB 0 through AB 5 in Figure 2) contains various pieces of information that tell the decoder how to decode the audio. These are bit fields that are extracted M bits at a time, where M is 0 to 16. MMX technology can be used to perform bit extraction [4], so we can efficiently parse the bit stream. From this information, we obtain the transform coefficients for the synthesis filter bank (TDAC transform).

Transform Coefficients Extraction

The audio block contains the information required to obtain the transform coefficients that will be sent to the synthesis filter bank. In Dolby Digital, the bit allocation, i.e., the number of bits used to represent a particular mantissa, is derived from the exponents (the spectral envelope). The mantissas are de-quantized and combined with the exponents in the denormalization process to create the transform coefficient values.

TDAC Transform

The Time Domain Aliasing Cancellation (TDAC) transform [5] converts the spectral information back to time domain, pulse-code modulated (PCM) samples. The TDAC provides perfect reconstruction (in the absence of quantization or other noise) and is critically sampled.

The TDAC transform is implemented as two DCT twiddle stages with an inverse Fast Fourier Transform (iFFT) in the middle [6]. A block diagram of this implementation of the TDAC transform is shown in Figure 3.

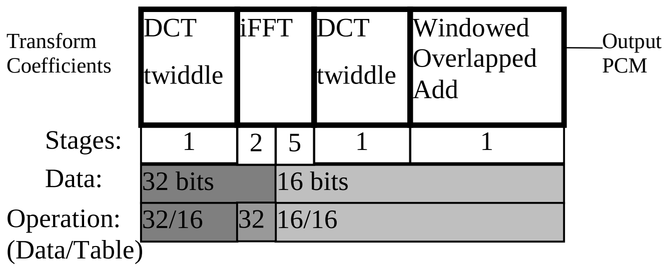

Figure 3. TDAC Transform Implementation

In our implementation, we created coefficient values with 24-bits of accuracy that are stored in 32-bit values. 24 bits of accuracy was chosen to prevent overflow during the intermediate denormalization and scaling processes. This 32-bit number was used in the first three stages of the TDAC transform. After the first two stages of the iFFT, the value was rounded to 16 bits of accuracy. The remainder of the operations were performed using pass-to-pass representations of 16 bits. MMX technology provides multiply accumulations to 32 bits, therefore many intermediate values were 32 bits.

The sine, cosine, and windowing values required in the TDAC transform were implemented via 16-bit lookup tables. Since these values are full-scale, 16 bits was sufficient for our needs. Errors introduced by imprecise coefficients are negligible compared to roundoff errors [7,8]. The technique of 32-bit data and 16-bit lookup tables has been shown to provide high-quality audio decoding [9].

Quantization errors introduced early in the transform process manifest themselves as tones in the output. Tonal noise is highly objectionable [10]. Output noise, if it must be present, should be broad-band or "white" noise. Therefore, the goal was to significantly reduce the peak spectral error. In a mixed-precision implementation, the question is how far into the TDAC transform do we need to carry 32 bits? In other words, where can we switch to 16 bits? Under subjective listening tests, we decided that performing the first three stages in 32 bits and the remainder in 16 bits reduced the tonal noise to a level of acceptability (see Figure 3). This also resulted in the decoder passing the measurement tests.

Multiplication in the MMX instruction set is 16 bits by 16 bits, yielding a 32-bit result. A 16-bit by 31-bit multiply is also possible in software, at a cost of at least five instructions and a pipelined output of two clocks per result [11]. Minimizing the number of 16-by-31 bit multiplies was important. It was discovered that the first two stages of the Decimation in Time (DIT) FFT contain only trivial coefficients, i.e., -1 and +1. This allowed these stages to be performed using only add and subtract instructions (no table lookup operations). These 32-bit operations are available in the MMX instruction set. This optimization allowed us to only use the more computationally intensive 16/31 bit operations only on the first DCT twiddle stage. The first two stages of the iFFT were performed with 32-bit adds and subtracts, which are efficient in the MMX instruction set.

The Windowed-Overlapped-Add (WOLA) block also fits well into the MMX instruction set. To perform the WOLA, the current and previous output arrays from the last DCT twiddle stage are windowed and then added together [5]. The windowing and addition operations were implemented as two 16-bit by 16-bit multiplies (the windowing) and then added as 32-bit quantities. This is provided by the PMADDWD instruction. The 32-bit results were then rounded to 16 bits for the output.

Mantissa Dithering

Dithering is required in a Dolby Digital decoder. How dithering is used by a decoder is determined by the Dolby Digital encoder used to create the frame. Dithering is used when the encoder determines that a transform mantissa doesn't get any bits (only an exponent) *and* that it is best to dither the mantissa (as opposed to having a mantissa of zero). This is implemented as a pseudo-random number generator that is random to 14 bits (the Dolby Digital specification states that the random number generator must be random to 8 bits or greater [3], so we exceed that specification). The calculation is given in Listing 1.

Listing 1. Dither Generation

C code:

```
x(t) = (x(t-1) * 47989) & 0xffff;
```

MMX Technology Assembly Code:

```
// dither multiplier value is linear
// congruential multiplier ^ 4,
// i.e. 0x4f31, packed 4 times

Quadword DithMultVal = 0x4f314f314f314f31;
```

```
// [63:48] = linear congruential multiplier ^ 4
// [47:32] = linear congruential multiplier ^ 3
// [31:16] = linear congruential multiplier ^ 2
// [15:0]  = linear congruential multiplier ^ 1
Quadword DithregInit = 0x4f31994d2379bb75;

Initialization:
        ;4 16-bit packed values
        MOVQ    MM0, DithregInit

Generation Loop:
        ; dither register * dither multiplier
        ; to get next set of values in dither
        ; register
        PMULLW  MM0, DithMultVal
        ;result is 4 16-bit values
        MOVQ    [result64], MM0

PMULLW has a latency of 3 but a throughput of
1.  This program can be pipelined to achieve
one result per clock written out to memory (for
example, on a Pentium processor PMULLW in the V
Pipe, MOVQ in the U Pipe).
```

Calculating four dither values with a single PMULLW instruction provides a high throughput for this part of the decoder. This instruction multiplies two 16-bit values and provides the lower 16 bits of the result (four of these are performed per instruction).

Downmixing

Dolby Digital can contain up to six audio channels: five full-bandwidth channels and a low-frequency effect (LFE) channel. This mode is often referred to as 5.1 channels, where 5 is the number of full-bandwidth channels and .1 is the LFE. The vast majority of PCs have only two audio output channels, so downmixing is often used. Also, for our two-channel downmix, the LFE is discarded. Downmixing is generally an additive process. Scaling (which is discussed below, see "Early Scaling") is also part of downmixing in Dolby Digital. It is used to set relative levels between downmixed channels. Since we perform it up front as part of the denormalization process, downmixing becomes additive. MMX technology provides SIMD addition, which speeds up downmixing.

Precision Enhancements

To increase the audio quality, some precision enhancements were made. Even though these techniques increased the processing requirement slightly, they added a significant quality improvement and were judged to be worth the additional overhead.

Rounding

Rounding is important to perform every time a higher-precision number is being converted to a lower-precision number (e.g., 32 to 16 bits). This is encountered often in multiplications in the MMX instruction set. For example, the PMADDWD instruction (packed multiply-accumulate) multiplies 16-bit numbers, yielding a 32-bit result. If this 32-bit result is to be converted to a 16-bit value, rounding should be used. Rounding can provide a significantly reduced error compared to truncation [7]. While the MMX instruction set does not provide a rounding mode, it is easy to accomplish in software. Listing 2 provides an example.

Listing 2. Rounding Using MMX Technology

```
// Round 2.30 number
// RoundVal is ½ LSB of 16-bit result

RoundVal = 0x0000400000004000;
pmaddwd mm6,mm5                 ;2.30 number
paddd   mm6,RoundVal            ;round
psrad   mm6,15                  ;2.30 to 1.15
```

Since the values are represented in two's complement, this technique works with both positive and negative numbers. In our Dolby Digital decoder, rounding was used extensively.

Gain Ranging

Dolby Digital provides Gain Ranging [3], which allows block scaling for low-level signals. This enhances the dynamic range of the decoder and was used in our implementation. Gain Ranging can contribute to noise modulation as the gain ranging levels are crossed. For our decoder, we decided the benefit of the additional dynamic range outweighed the potential of discontinuous noise modulation.

Additional performance enhancements were made that are general to the implementation of a Dolby Digital decoder on a PC. These are included here, even though they are not unique to optimizations that utilize MMX technology.

Additional Performance Enhancements

Frequency Domain Downmixing

Since the TDAC transform is a linear process, downmixing can be accomplished in the frequency domain. This reduces the number of transforms from the number of input channels from the Dolby Digital stream (2 to 5) to the number of output channels (2). However, the transform block sizes in Dolby Digital can change from 512 to 256 in the presence of transients [3]. It is not possible to downmix in the frequency domain for differing block sizes, so in this case an additional downmix stage is required after the TDAC transform to perform the remainder of the downmix in the time domain. The transform coefficients are contained in 32-bits. Using the 32-bit adds in the MMX instruction set provides an efficient downmix.

Early Scaling

There are several factors in the scale factor of a particular channel: Dynamic Range Control, Gain Ranging, and Downmix Scaling. We found it computationally beneficial to perform this operation during denormalization, essentially combining scaling and denormalization into one operation. This is performed by adjusting all of the exponents and mantissas by a particular amount. We stored the exponents as 8-bit quantities (the range is only 5 bits in Dolby Digital) and used MMX technology 8-bit add instructions (PADDB) to scale 8 exponents at a time. The unpack instruction (PUNPCKLBW) was used to efficiently replicate the 8-bit scale value eight times across the 64-bit register.

When the values are scaled up front, then downmixing becomes a simple addition as opposed to a multiplication by a constant. Since the transform coefficients are represented in 32 bits, downmixing in the frequency domain is performed by 32-bit adds using the packed add (PADDD) instruction. This avoids 32-bit multiplies.

Exponent and Bit Allocation Reuse

A Dolby Digital stream only has exponents in an audio block when the encoder determines that they have changed enough to be resent. This is called exponent reuse. Therefore, if exponent reuse is in effect, it is more processor-efficient to save the exponents in an array and use the values from the array (as opposed to re-extracting the bits from the bit stream).

The bit allocation information is derived from the exponents. Therefore if exponent reuse is in effect, bit allocation may be also (depending on new bit allocation information, SNR offset information, delta bit allocation information and coupling information - see [3] for details). Since recalculating the bit allocation information is computationally expensive, the bit allocation information should be saved in an array and reused if possible. This does not benefit from MMX technology per se, but shows the advantages of decoding on a system that has a relatively large amount of cache memory as opposed to a DSP that may have to recalculate these values since it does not have sufficient memory for all of these arrays.

Results

Compared to an optimized version that does not use MMX technology, the processor speedup is about 1.5X for a two-channel, surround-compatible (also known as LtRt) downmix from 5.1 channel source material. For 5.1 channels of output, the speedup increases to about 1.8X. Typically, two TDAC transforms are performed for a two-channel downmix, and six are required for a full 5.1 channels of output. The greater speedup is due to the fact that the TDAC transform benefits greatly from MMX technology and the increased number of TDAC transforms performed for 5.1 channels (versus a two-channel downmix). One caveat is that six channels of audio output is not common on today's mainstream PC. However, sound cards with four channels of discrete audio output are on the market today, so six channels may become available in the future via analog outputs or the 1394 high-speed serial bus.

Intel's Dolby Digital decoder provides significantly better audio quality than a 16-bit only approach, while offering an efficient implementation. The included audio clips contrast the 16-bit only approach with the enhanced approach. Note that these are very low-level signals (you may have to increase the volume to hear them).

noisfl16.wav

Low-level noise decoded by a simple 16-bit implementation. Notice the tonal artifacts. (Sound file is only availiable in online HTML version.)

noisfl_m.wav

Low-level noise decoded by Intel's mixed 16/32-bit implementation. The noise is lower and broad-band (white). (Sound file is only available in online HTML version.)

Intel's Dolby Digital decoder compares favorably with floating-point based implementations. Typically Intel's decoder has about 5 to 10 dB of additional noise as compared to a floating-point implementation. The improvement over a simple 16-bit truncation model is approximately 5 to 15 dB, depending on the program material. The most striking improvement is the reduction in peak spectral error, or the "tonality."

Figures 4 through 10 show how Intel's decoder compares to the 16-bit truncation model and floating-point reference.

Figures 4 through 7 show a spectral plot of a 200 Hz sine wave at -60 dB. Figure 4 is a composite of Figures 5 through 7. These are separated out since, in the composite, it is difficult to distinguish between the three plots. This illustrates the peak spectral error (graphical peaks) in the 300 to 20 kHz region. These peaks show the presence of tonal noise. The 16-bit truncation decoder has by far the worst peaks, as high as -105 dB. The MMX technology decoder reduces these peaks by 13 dB to -118 dB.

Figure 8 shows the Total Harmonic Distortion (THD) vs. Frequency. The THD vs. Frequency is improved by about 10 dB over the 16-bit truncation decoder.

Figure 9 is the noise modulation plot. This is a plot of the output noise in a third octave band at 4 kHz as a function of the input level of a 41 Hz sinusoid decremented from 0 dBFS to -120 dBFS. The improved (lowered) noise level is between 15 dB for high-level signals and 5 dB for low-level signals.

Figure 10 is a noise plot of a 4 kHz sine wave reduced in level 1 dB per second from 0 dBFS to -120 dBFS, with the sine wave removed via a notch filter. This shows that the noise for a full-level signal is still small (-78 dB), going to -88 dB for a medium- to low-level signal. This is approximately a 12 dB improvement for high-level signals and approximately a 6 dB improvement for low-level signals.

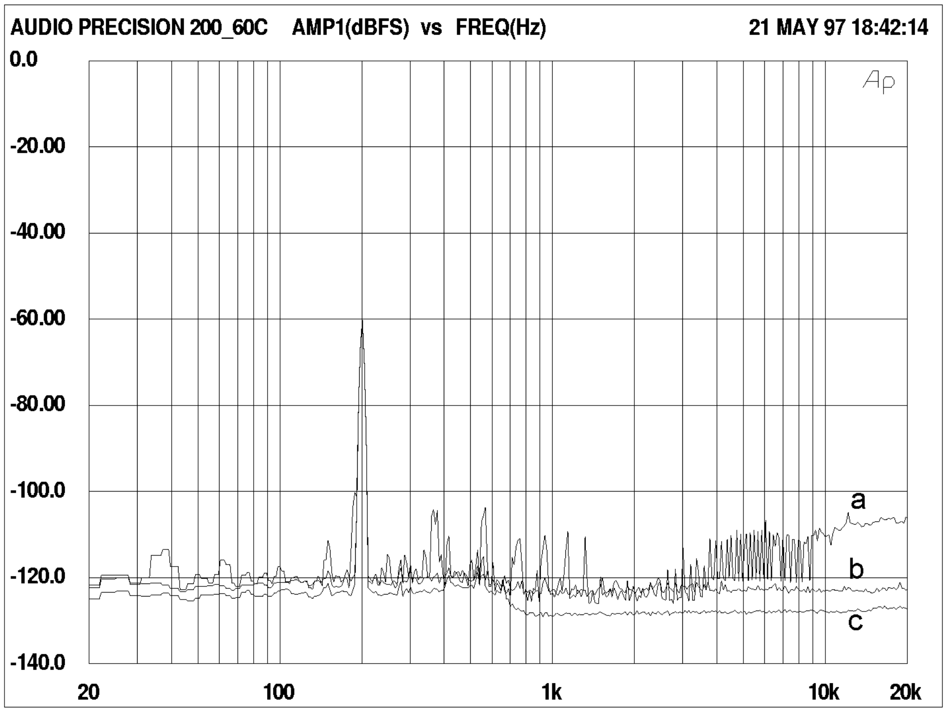

Figure 4. 200 Hz at -60 dB. a) 16-Bit Truncation, b) MMX Technology, c) Dolby Reference Decoder

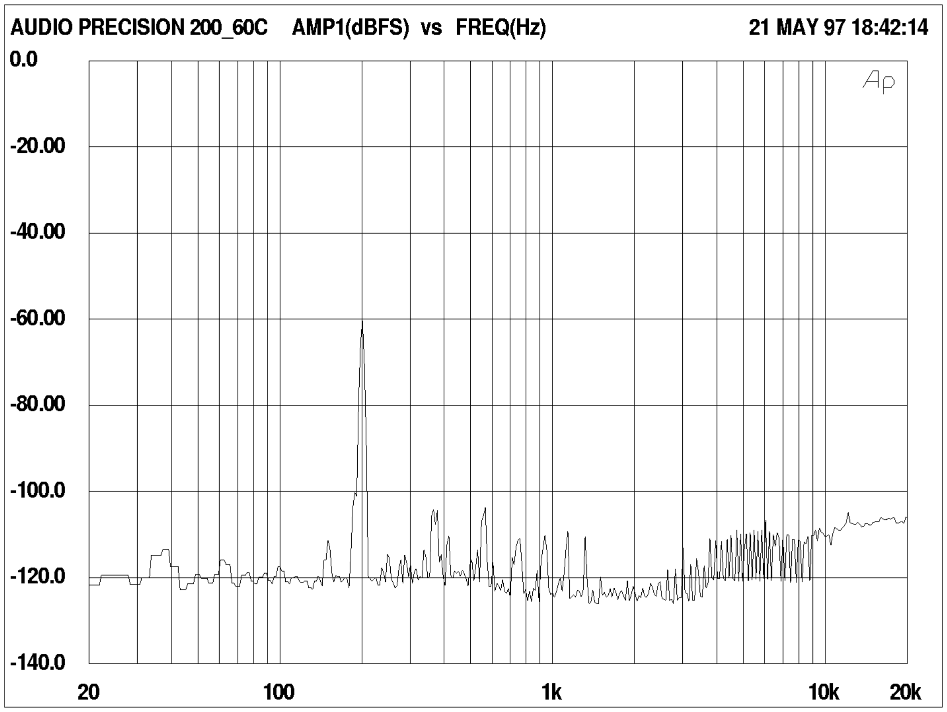

Figure 5. 200 Hz at -60 dB. 16-Bit Truncation Decoder

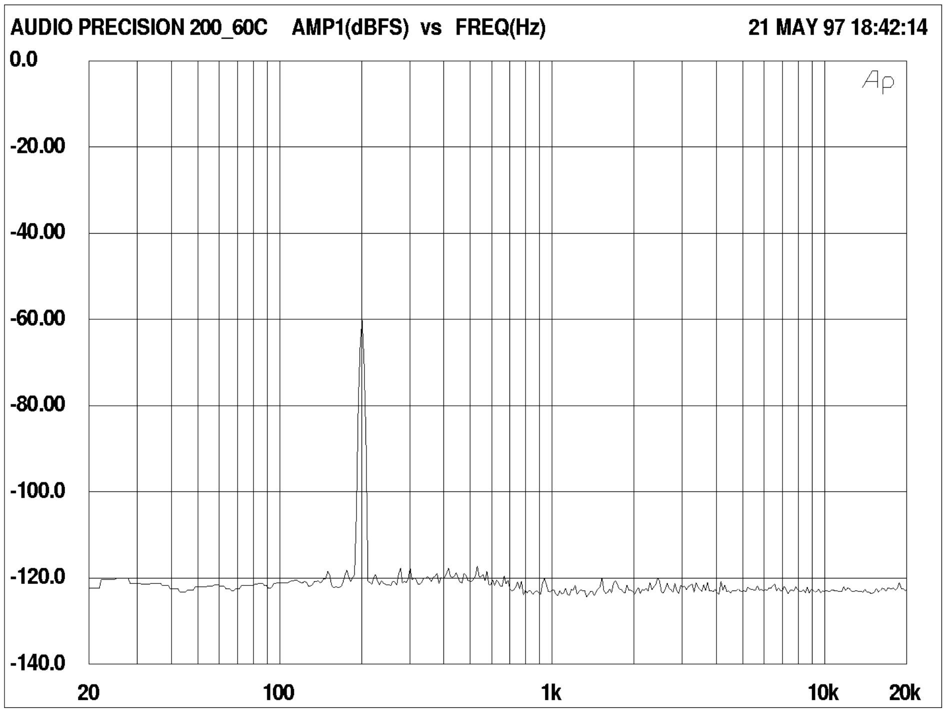

Figure 6. 200 Hz at -60 dB. MMX Technology Decoder

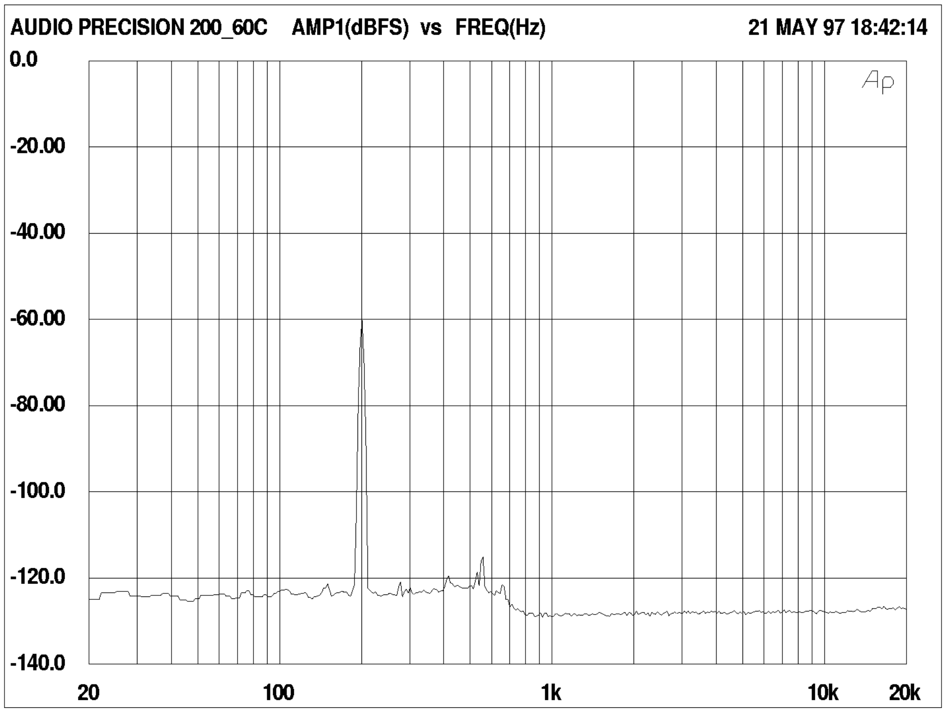

Figure 7. 200 Hz at -60 dB. Dolby Reference Decoder

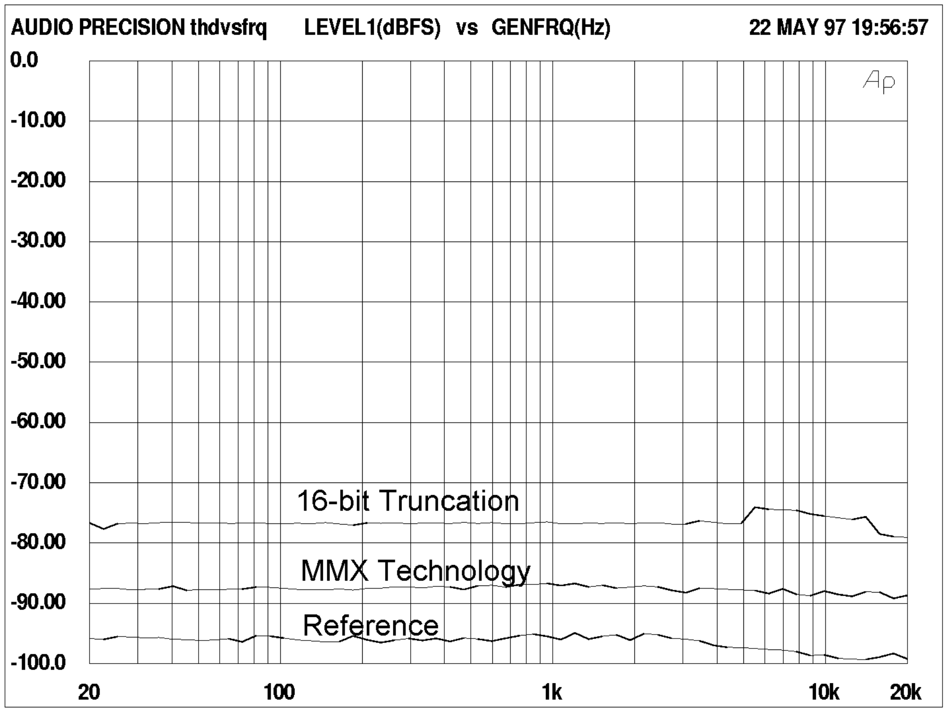

Figure 8. THD vs. Frequency

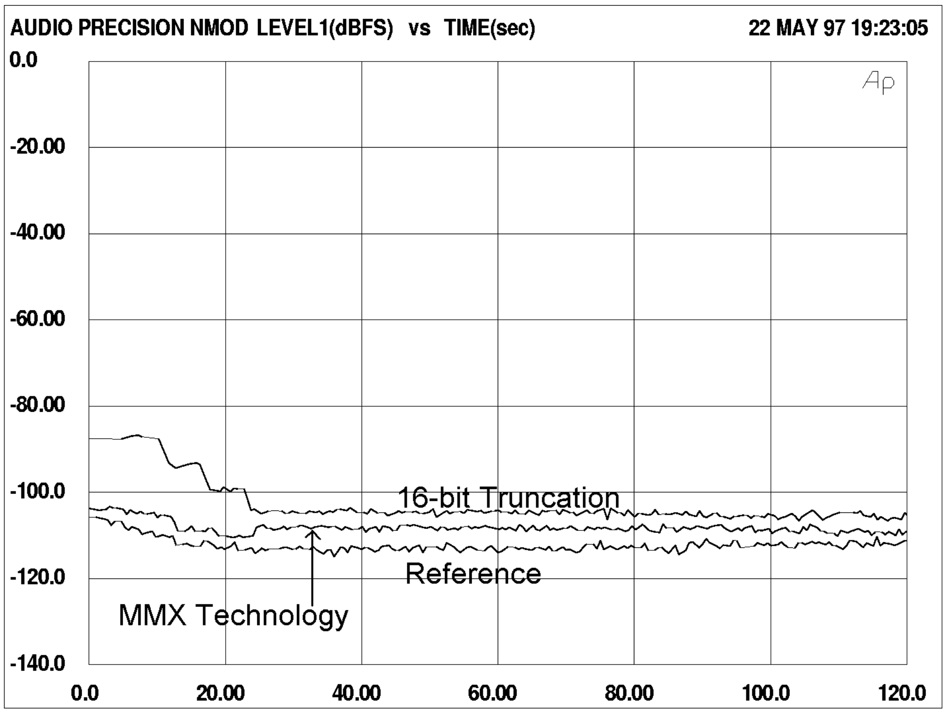

Figure 9. Noise Modulation at 4 kHz, 41 Hz Input

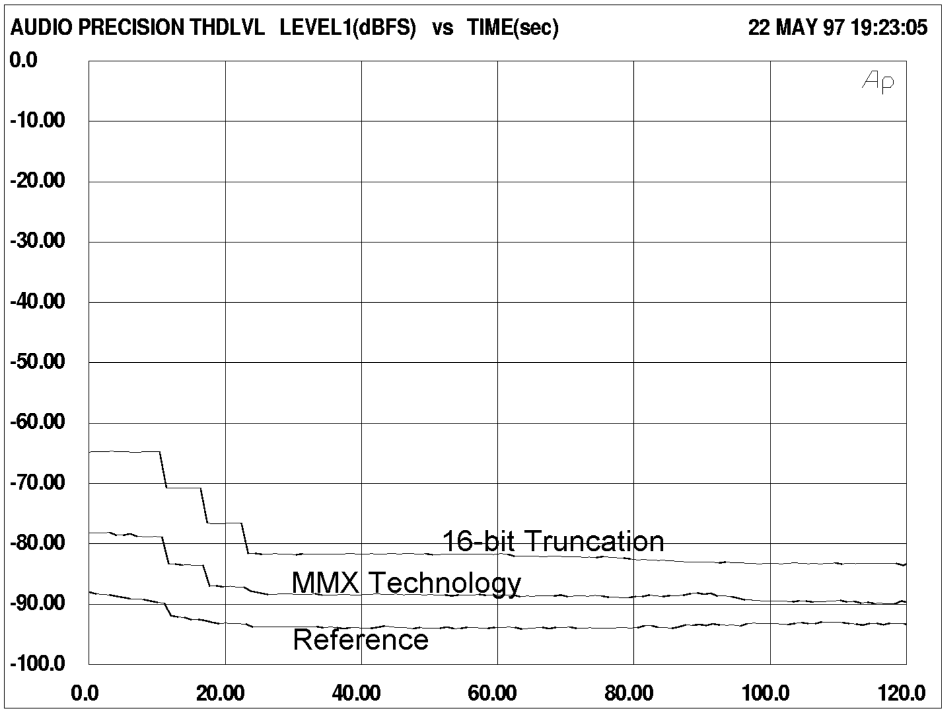

Figure 10. THD vs. Level, 4 kHz Input

Decoding a Dolby Digital stream consumes less than 8% of a Pentium® II processor running at 233 MHz. Figure 11 shows the processor requirements for several DVD audio tracks (5.1 channels, 384K bits/second, 48K samples/second, downmixed to LtRt, except for TWISTER which is two channels, 192K bits/second). Clearly, this is small enough to make software Dolby decoding quite feasible in real-world applications. The remaining 92% of the processor can be used for other things, such as a software MPEG 2 video decoder for a software DVD player.

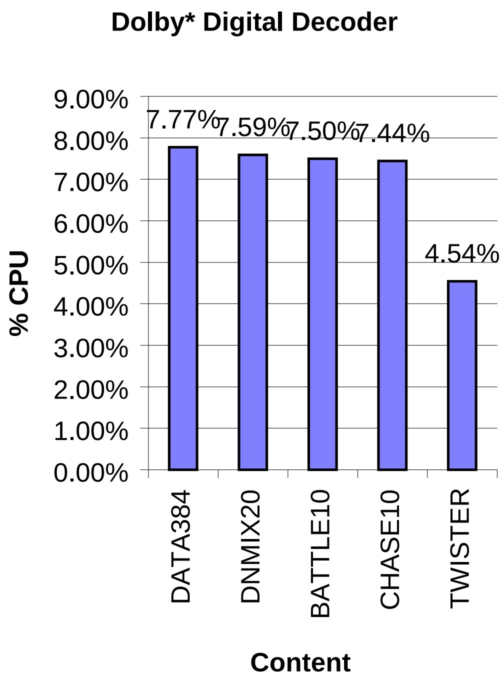

Figure 11. Processor Requirements. Note - DATA384 and DNMIX20 are test materials. BATTLE10 and CHASE10 are from the movie *Outbreak*. TWISTER is from the movie *Twister*.

Discussion

Making intelligent use of MMX technology requires a good understanding of the algorithm being coded. By understanding the strengths and flexibility of MMX technology, many clever techniques can be devised. While high-quality audio is a subjective term, we believe this decoder lives up to the name.

Table 1 shows the CPU breakdown for each part of the Dolby Digital decoder. After the data path has been sped up by MMX technology, the Bit Unpacking section becomes the next major consumer of the CPU. This is mainly due to the sequential nature of extracting variable-length bit fields from the bit stream.

Table 1. CPU Breakdown

Processing Block	% of Full Decoder
Bit Unpacking	28.3
TDAC/WOLA/Downmix	27.7
Scaling/Denormalization	27.2
Bit Allocation	10.2
Miscellaneous	6.6

Based on measurements (see Figure 10), the Intel decoder has a Signal-to-Noise Ratio (SNR) for a full-scale signal of about 78 dB. This compares reasonably well to the instantaneous sensitivity of the ear of about 85 dB [2]. The Dynamic Range (maximum output level vs. noise floor for a low-level signal) is about 88 dB. This compares reasonably well to a consumer CD player, which is typically at about 95 dB.

Conclusion

Intel's Dolby Digital decoder provides a processor-efficient implementation that meets a high-quality standard. By offering this decoder as a baseline capability on PCs with MMX technology, decoding and playback of compressed audio is possible with no additional hardware cost. The low processor usage allows additional features such as software video decoding and audio enhancement to occur concurrently.

Acknowledgment

The writers thank Eric Benjamin, Charles Seagrave, Uttam Suri, and Steve Vernon of Dolby Laboratories for their suggestions on quality improvement and the help they provided in the analysis of the Intel decoder.

Authors

James Abel is currently a Software Development Engineering Manager in the Desktop Performance Lab at Intel Corporation in Chandler, Arizona. James obtained a Bachelor's Degree in Engineering from Bradley University in Peoria, Illinois in 1983 and a Master's Degree in Computer Science from Arizona State University in 1991. His interests include signal processing, computer architectures, software tools, and audio algorithms. James' email address is jabel@inside.intel.com.

Mike Julier is currently a Sr. Software Development Engineer in the Desktop Performance Lab at Intel Corporation in Chandler, Arizona. He holds a BS in Computer Engineering from the University of Michigan, Ann Arbor. Mike's technical interests include performance optimizations of code from C++ to assembly, 3D graphics, and ISAs. Mike's email address is Michael_A_Julier@ccm.sc.intel.com.

References

[1] Fielder, L., Bosi, M., Davidson, G., Davis, M., Todd, C., and Vernon, S., "AC-2 and AC-3: Low-Complexity Transform-Based Audio Coding," Collected Papers on Digital Audio Bit-Rate Reduction, Audio Engineering Society, New York, New York, pp. 54-72.

[2] Harris, S., "Understanding, enhancing, and measuring PC-audio quality," EDN, Vol. 42, Number 8, April 10, 1997, p. 173.

[3] Advanced Television Systems Committee, "Digital Audio Compression Standard (AC-3)," Revision A/52, 20 December 1995.

[4] "Using MMX™ Instructions to Get Bits From a Data Stream," http://developer.intel.com/drg/mmx/appnotes.

[5] Princen, J. and Bradley, A., "Analysis/Synthesis Filter Back Design Based on Time Domain Aliasing Cancellation," IEEE Transactions on ASSP, Vol. 34, pp. 1153-1161, October 1986.

[6] Sevic, D., Popovic, M., "A New Efficient Implementation of the Oddly Stacked Princen-Bradley Filter Bank," IEEE Signal Processing Letters, Vol. 1, No. 11, pp. 166-168, November 1994.

[7] Thong, T., "Fixed-Point Fast Fourier Transform Error Analysis," IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-24, No. 6, December 1976.

[8] Weinstein, C., "Quantization Effects in Digital Filters," MIT Lincoln Laboratories Technical Report 468, ASTIA Doc. DDC AD-706862, November 21, 1969.

[9] Davidson, G., Anderson, W., Lovrich, A., "A Low-Cost Adaptive Transform Decoder Implementation for High-Quality Audio," IEEE International Conference on

Acoustics, Speech, and Signal Processing, March 23-26, 1992.

[10] Dolby Laboratories, “Test Procedure for AC-3 Decoders Using Test Files,” Dolby Document S96/11283.

[11] “Using MMX™ Instructions to Perform 16-Bit x 31-Bit Multiplication,” http://developer.intel.com/drg/mmx/appnotes.
