3

Application Programming Model

CHAPTER 3
APPLICATION PROGRAMMING MODEL

This chapter describes the application programming environment as seen by compiler writers and assembly-language programmers. It also describes the architectural features which directly affect applications.

3.1. DATA FORMATS

3.1.1. Memory Data Formats

The Intel Architecture MMX™ technology introduces new packed data types, each 64 bits long. The data elements can be:

- eight packed, consecutive 8-bit bytes
- four packed, consecutive 16-bit words
- two packed, consecutive 32-bit doublewords

The 64 bits are numbered 0 through 63. Bit 0 is the least significant bit (LSB), and bit 63 is the most significant bit (MSB).

The low-order bits are the lower part of the data element and the high-order bits are the upper part of the data element. For example, a word contains 16 bits numbered 0 through 15, the byte containing bits 0-7 of the word is called the low byte, and the byte containing bits 8-15 is called the high byte.

Bytes in a multi-byte format have consecutive memory addresses. The ordering is always little endian. That is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

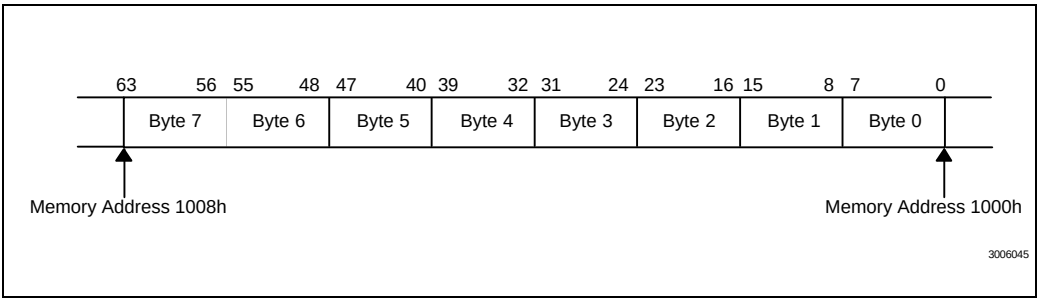

Figure 3-1. Eight Packed Bytes in Memory (at address 1000H)

3.1.2. IA MMX™ Register Data Formats

Values in IA MMX registers have the same format as a 64-bit quantity in memory. MMX registers have two data access modes: 64-bit access mode and 32-bit access mode.

The 64-bit access mode is used for 64-bit memory access, 64-bit transfer between MMX registers, all pack, logical and arithmetic instructions, and some unpack instructions.

The 32-bit access mode is used for 32-bit memory access, 32-bit transfer between integer registers and MMX registers, and some unpack instructions.

3.1.3. IA MMX™ Instructions and the Floating-Point Tag Word

After each MMX instruction, the entire floating-point tag word is set to Valid (00s). The Empty MMX State (EMMS) instruction sets the entire floating-point tag word to Empty (11s).

Section 4.3.2. describes the effects of floating-point and MMX instructions on the floating-point tag word. For details on floating-point tag word, refer to the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 6.2.1.4.

3.2. PREFIXES

Table 3-1 details the effect of a prefix on IA MMX instructions.

Table 3-1. IA MMX™ Instruction Behavior with Prefixes Used by Application Programs

Prefix Type	The Effect of Prefix on IA MMX™ Instructions
Address size (67H)	Affects IA MMX instructions with a memory operand. Ignored by IA MMX instructions without a memory operand.
Operand size (66H)	Ignored.
Segment override	Affects IA MMX instructions with a memory operand. Ignored by IA MMX instructions without a memory operand.
Repeat	Ignored.
Lock (F0H)	Generates an invalid opcode exception.

See the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 3.4. for information related to prefixes.

3.3. WRITING APPLICATIONS WITH IA MMX™ CODE

3.3.1. Detecting IA MMX™ Technology Existence Using the CPUID Instruction

Use the CPUID instruction to determine whether the processor supports the IA MMX instruction set (refer to the *Pentium® Processor Family Developer's Manual*, Volume 3, Chapter 25, for more detail on the CPUID instruction). When the IA MMX technology support is detected by the CPUID instruction, it is signaled by setting bit 23 (IA MMX technology bit) in the feature flags to 1. In general, two versions of the routine can be created: one with scalar instructions and one with MMX instructions. The application will call the appropriate routine depending on the results of the CPUID instruction. If MMX technology support is detected, then the MMX routine is called; if no support for the MMX technology exists, the application calls the scalar routine.

NOTE

The CPUID instruction will continue to report the existence of the IA MMX technology if the CR0.EM bit is set (which signifies that the CPU is configured to generate exception Int 7 that can be used to emulate floating

point instructions). In this case, executing an MMX instruction results in an invalid opcode exception.

Example 3-1 illustrates how to use the CPUID instruction. This example does not represent the entire CPUID sequence, but shows the portion used for IA MMX technology detection.

Example 3-1. Partial sequence of IA MMX™ technology detection by CPUID

```
...                             ; identify existence of CPUID instruction
...
...                             ; identify Intel processor
....
mov     EAX, 1                  ; request for feature flags
CPUID                           ; 0Fh, 0A2h CPUID instruction
test    EDX, 00800000h          ; Is IA MMX technology bit (Bit 23 of EDX) in feature flags set?
jnz     MMX_Technology_Found
```

3.3.2. The EMMS Instruction

When integrating the MMX routine into an application running under an existing operating system (OS), programmers need to take special precautions, similar to those when writing floating-point (FP) code.

When an MMX instruction executes, the floating-point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating-point tag word as empty. Therefore, it is imperative to use the EMMS instruction at the end of every MMX routine.

The EMMS instruction must be used in each of the following cases:

- Application utilizing FP instructions calls an MMX technology library/DLL
- Application utilizing MMX instructions calls a FP library/DLL
- Switch between MMX code in a task/thread and other tasks/threads in cooperative operating systems.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating-point exception event may be generated.

- A "soft exception" may occur. In this case floating-point code continues to execute, but generates incorrect results. This happens when the floating-point exceptions are masked and no visible exceptions occur. The internal exception handler (microcode, not user visible) loads a NaN (Not a Number) with an exponent of 11..11B onto the floating-point stack. The NaN is used for further calculations, yielding incorrect results.
- A potential error may occur only if the operating system does NOT manage floating-point context across task switches. These operating systems are usually cooperative operating systems. It is imperative that the EMMS instruction execute at the end of all the MMX routines that may enable a task switch immediately after they end execution (explicit yield API or implicit yield API).

3.3.3. Interfacing with IA MMX™ Technology Procedures and Functions

The MMX technology enables direct access to all the MMX registers. This means that all existing interface conventions that apply to the use of other general registers such as EAX, EBX will also apply to the MMX register usage.

An efficient interface might pass parameters and return values via the pre-defined MMX registers, or a combination of memory locations (via the stack) and MMX registers. This interface would have to be written in assembly language since passing parameters through MMX registers is not currently supported by any existing C compilers. Do not use the EMMS instruction when the interface to the MMX code has been defined to retain values in the MMX register.

If a high-level language, such as C, is used, the data types could be defined as a 64-bit structure with packed data types.

When implementing usage of IA MMX instructions in high level languages other approaches can be taken, such as:

- Passing MMX type parameters to a procedure by passing a pointer to a structure via the integer stack.
- Returning a value from a function by returning the pointer to a structure.

3.3.4. Writing Code with IA MMX™ and Floating-Point Instructions

The MMX technology aliases the MMX registers on the floating-point registers. The main reason for this is to enable MMX technology to be fully compatible and transparent to existing software environments (operating systems and applications). This way operating

systems will be able to include new applications and drivers that use the IA MMX technology.

An application can contain both floating-point and MMX code. However, the user is discouraged from causing frequent transitions between MMX and floating-point instructions by mixing MMX code and floating-point code.

3.3.4.1. RECOMMENDATIONS AND GUIDELINES

Do not mix MMX code and floating-point code at the instruction level for the following reasons:

- The TOS (top of stack) value of the floating-point status word is set to 0 after each MMX instruction. This means that the floating-point code loses its pointer to its floating-point registers if the code mixes MMX instructions within a floating-point routine.
- An MMX instruction write to an MMX 64-bit register writes ones (11s) to the exponent part of the corresponding floating-point register.
- Floating-point code that uses register contents that were generated by the MMX instructions may cause floating-point exceptions or incorrect results. These floating-point exceptions are related to undefined floating-point values and floating-point stack usage.
- All MMX instructions (except EMMS) set the entire tag word to the valid state (00s in all tag fields) without preserving the previous floating-point state.
- Frequent transitions between the MMX and floating-point instructions may result in significant performance degradation in some implementations.

If the application contains floating-point and MMX instructions, follow these guidelines:

- Partition the MMX technology module and the floating-point module into separate instruction streams (separate loops or subroutines) so that they contain only instructions of one type.
- Do not rely on register contents across transitions.
- When the MMX state is not required, empty the MMX state using the EMMS instruction.
- Exit the floating-point code section with an empty stack.

Example 3-2. Floating-point and MMX™ Code

```
FP_code:
                ..
                ..          (*leave the FP stack empty*)
MMX_code:
                ..
                EMMS        (*mark the FP tag word as empty*)
FP_code 1:
                ..
                ..          (*leave the FP stack empty*)
```

3.3.5. Multitasking Operating System Environmen t

An application needs to identify the nature of the multitasking operating system on which it runs. Each task retains its own state which must be saved when a task switch occurs. The processor state (context) consists of the integer registers and floating-point and MMX registers.

Operating systems can be classified into two types:

- Cooperative multitasking operating system
- Preemptive multitasking operating system

The behavior of the two operating system types in context switching is described in Section 4.1.1.

3.3.5.1. COOPERATIVE MULTITASKING OPERATING SYSTEM

Cooperative multitasking operating systems do not save the FP or MMX state when performing a context switch. Therefore, the application needs to save the relevant state before relinquishing direct or indirect control to the operating system.

3.3.5.2. PREEMPTIVE MULTITASKING OPERATING SYSTEM

Preemptive multitasking operating systems are responsible for saving and restoring the FP and MMX state when performing a context switch. Therefore, the application does not have to save or restore the FP and MMX state.

3.3.6. Exception Handling in IA MMX™ Application Code

MMX instructions generate the same type of memory-access exceptions as other Intel Architecture instructions. Some examples are: page fault, segment not present, and limit violations. Existing exception handlers can handle these types of exceptions. They do not have to be modified.

Unless there is a pending floating-point exception, MMX instructions do not generate numeric exceptions. Therefore, there is no need to modify existing exception handlers or add new ones.

If a floating-point exception is pending, the subsequent MMX instruction generates a numeric error exception (Int 16 and/or FERR#). The MMX instruction resumes execution upon return from the exception handler.

3.3.7. Register Mapping

The IA MMX registers and their tags are mapped to physical locations of the floating-point registers and their tags. Register aliasing and mapping is described in more detail in Section 4.3.1.