AMD

AMD-K6®

Processor

Code
Optimization

Application Note

Trademarks

AMD, the AMD logo, K6, 3DNow!, and combinations thereof, K86, Super7, and AMD-K5 are trademarks, and RISC86 and AMD-K6 are registered trademarks of Advanced Micro Devices, Inc.

MMX is a trademark and Pentium is a registered trademark of the Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Contents

List of Figures

List of Tables

Revision History

Date	Rev	Description
Feb 1998	A	Initial Release
May 1998	B	Added IN and OUT instructions to Table 6, "Integer Instructions," on page 29.
May 1998	B	Clarified address modes on page 68 and page 82.
August 1999	C	Changed title and Introduction to reflect that the information in this document applies to the AMD-K6® family of processors - mainly to the AMD-K6-2 and AMD-K6-III processors.
August 1999	C	Revised address mode information on page 76.
August 1999	C	Revised examples in "Division and Square Root" on page 90.

1

Introduction

Purpose

The AMD K86™ family of x86 processors can efficiently execute code written for previous-generation x86 processors. However, to get the highest performance from the unique microarchitecture of the AMD-K6® family of processors, certain code optimization techniques should be applied.

This document contains information to assist programmers in creating optimized code for the AMD-K6 family. This document is targeted at compiler/assembler designers and assembly language programmers writing high-performance code sequences. It is assumed that the reader possesses an in-depth knowledge of the x86 architecture.

The information in this application note pertains to the AMD-K6 family of processors – information specific to the AMD-K6-2 processor Model 8 and AMD-K6-III processor Model 9 is noted. For information about the recognition of processor model numbers, see the *AMD Processor Recognition Application Note*, order# 20734.

AMD-K6® Family of Processors

Processors in the AMD-K6 family use a decoupled instruction decode and superscalar execution microarchitecture, including state-of-the-art RISC design techniques, to deliver sixth-generation performance with x86 binary software compatibility. An x86 binary-compatible processor implements the industry-standard x86 instruction set by decoding and executing the x86 instruction set as its native mode of operation. Only this native mode permits delivery of maximum performance when running PC software.

AMD-K6®-2 and AMD-K6®-III Processors

The AMD-K6-2 and AMD-K6-III processors (hereafter both are referred to as the processor) bring superscalar RISC performance to desktop systems running industry-standard x86 software. The processor implements advanced design techniques such as:

■ Instruction pre-decoding

■ Multiple x86 opcode decoding

■ Single-cycle internal RISC operations

■ Multiple parallel execution units

■ Out-of-order execution

■ Data-forwarding

■ Register renaming

■ Dynamic branch prediction

The processor is capable of issuing, executing, and retiring multiple x86 instructions per cycle, resulting in superior scaleable performance.

Although the processor is capable of extracting code parallelism out of off-the-shelf, commercially available x86 software, specific code optimizations for the processor can result in significantly higher delivered performance. This document describes the RISC86® microarchitecture in the processor and makes recommendations for optimizing

execution of x86 software on the processor. The coding techniques for achieving peak performance on the processor include, but are not limited to, those recommended for the Pentium®, Pentium II, and Pentium Pro processors. However, many of these optimizations are not necessary for the processor to achieve maximum performance. For example, due to more flexible pipeline control in the AMD-K6 microarchitecture, the processor is less sensitive to instruction selection and the scheduling of code. This flexibility is one of the distinct advantages of the AMD-K6 processor microarchitecture.

In addition to the ability to execute MMX™ instructions, the processor includes the implementation of the 3DNow!™ instruction set. 3DNow! technology was created based on suggestions from leading graphics and software vendors. Utilizing a data format and single instruction multiple data (SIMD) operations based on the MMX instruction model, the processor can produce up to four, 32-bit, single-precision floating-point results per clock cycle. 3DNow! technology also includes new integer multimedia instructions, a new instruction to allow the prefetching of data under software control, and a faster enter/exit multimedia-state instruction.

The 3DNow! units provide support for high-performance, floating-point vector operations, which can replace x87 instructions and enhance the performance of 3D graphics and other floating-point-intensive applications. The complete multimedia processing unit in the processor combines existing MMX instructions with the new 3DNow! instructions. The 3DNow! instructions share the use of the MMX registers with the multimedia unit. By mixing 3DNow! instructions with MMX instructions, it now becomes possible to write x86 programs containing both MMX integer and floating-point instructions without a performance penalty that would have been incurred if MMX and x87 floating-point instructions were intermixed. All these improvements have been carefully designed to bring a better multimedia experience to mainstream PC users while maintaining backwards compatibility with all existing x86 software.

2

RISC86® Microarchitecture

Overview

When discussing processor design, it is important to understand the terms *architecture*, *microarchitecture*, and *design implementation*. The term *architecture* refers to the instruction set and features of a processor that are visible to software programs running on the processor. The architecture determines what software the processor can run. The architecture of the AMD-K6 processor is the industry-standard x86 instruction set.

The term *microarchitecture* refers to the design techniques used in the processor to reach the target cost, performance, and functionality goals. The AMD-K6 processor is based on a sophisticated RISC core known as the Enhanced RISC86 microarchitecture. The Enhanced RISC86 microarchitecture is an advanced decoupled decode/execution design approach that enables industry-leading performance for x86-based software.

The term *design implementation* refers to the actual logic and circuit designs from which the processor is created according to the microarchitecture specifications.

Enhanced RISC86® Microarchitecture

The Enhanced RISC86 microarchitecture defines the characteristics of the AMD-K6 family of processors. The innovative RISC86 microarchitecture approach implements the x86 instruction set by internally translating x86 instructions into RISC86 operations. These RISC86 operations were specially designed to include direct support for the x86 instruction set while observing the RISC performance principles of fixed-length encoding, regularized instruction fields, and a large register set. The Enhanced RISC86 microarchitecture used in the AMD-K6 processor enables higher processor core performance and promotes straightforward extensions in future designs. Instead of directly executing complex x86 instructions, which have lengths of 1 to 15 bytes, the AMD-K6 processor executes the simpler fixed-length RISC86 operations, while maintaining the instruction coding efficiencies found in x86 programs.

The AMD-K6 processor includes parallel instruction decoders, a centralized RISC86 operation scheduler, and several execution resources that support superscalar execution—multiple decode, execution, and retirement—of x86 instructions. These elements are packed into an aggressive and highly efficient six-stage processing pipeline.

Decoding of the x86 instructions into RISC86 operations begins when the on-chip level-one instruction cache is filled. Predecode logic determines the length of an x86 instruction on a byte-by-byte basis. This predecode information is stored along with the x86 instructions in a dedicated, level-one predecode cache to be used later by the decoders. The predecode data is essential to the ability of the short decoders to operate.

The AMD-K6 processor categorizes x86 instructions into three types of decodes—short, long, and vector. The decoders process either two short, one long, or one vector decode at a time. The three types of decodes have the following characteristics:

■ Short decodes—common x86 instructions less than or equal to 7 bytes in length that produce one or two RISC86 operations. The two short decoders can work in parallel, resulting in a maximum of four RISC86 operations per clock with no additional latency.

■ Long decodes—more complex and somewhat common x86 instructions less than or equal to 11 bytes in length that produce up to four RISC86 operations.

■ Vector decodes—complex x86 instructions requiring long sequences of RISC86 operations.

Short and long decodes are processed completely within the decoders. Vector decodes are started by the vector decoder with the generation of an initial set of four RISC86 operations, and then completed by fetching a sequence of additional operations from an on-chip ROM (at a rate of four operations per clock). RISC86 operations, whether produced by decoders or fetched from ROM, are then loaded into a buffer line in the centralized scheduler for dispatch to the execution units.

AMD-K6®-2 and AMD-K6®-III Processor-Specific Microarchitecture

The internal RISC86 instruction set consists of the following seven categories or types of operations (the execution unit that handles each type of operation is displayed in parenthesis):

■ Memory load operations (load)

■ Load immediate (instruction control unit)

■ Memory store operations (store)

■ Integer register operations (alu/alux)

■ MMX/3DNow! register operations (multimedia execution unit (meu))

■ x87 floating-point register operations (float)

■ Branch condition evaluations (branch)

The following example shows a series of x86 instructions and the corresponding decoded RISC86 operations.

```
x86 Instructions              RISC86 Operations

MOV   CX, [SP+4]  ────────→   Load
ADD   AX,BX       ────────→   Alu (Add)
CMP   CX,[AX]     ────────→   Load
                              Alu (Sub)
JZ    foo         ────────→   Branch
```

The MOV instruction converts to a RISC86 load operation that requires indirect data to be loaded from memory. The ADD instruction converts to an alu register operation that can be sent to either of the integer units. The CMP instruction converts into two RISC86 operations. The first RISC86 load

operation requires indirect data to be loaded from memory. That value is then compared (alu function) with CX.

Once the RISC86 operations are placed in the centralized scheduler buffer, they can be immediately issued to the appropriate execution pipeline. The processor contains ten execution pipelines—store, load, integer X ALU, integer Y ALU, MMX ALU (X), MMX ALU (Y), MMX/3DNow! multiplier, 3DNow! ALU, Floating-Point, and Branch. Figure 1 shows a block diagram of these units within the processor. The Register X and Y Functional Units contain several execution resources, which are described in Chapter 3 on page 11.
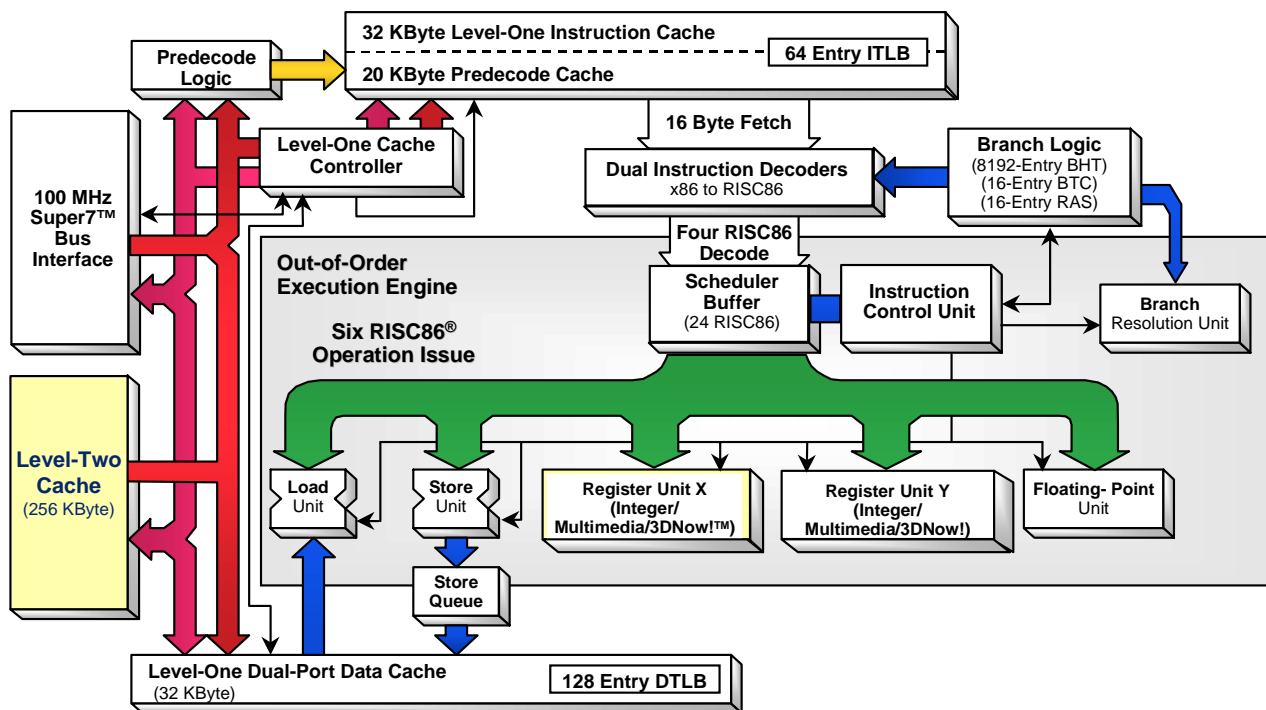
Figure 1.   AMD-K6®-III Processor Block Diagram

The centralized scheduler buffer, in conjunction with the instruction control unit (ICU), buffers and manages up to 24 RISC86 operations at a time (which equals up to 12 x86 instructions). This buffer size is matched to the processor's six-stage RISC86 pipeline and decode rate of four RISC86 operations per clock.

On every clock, the centralized scheduler buffer can accept up to four RISC86 operations from the decoders, issue up to six RISC86 operations to corresponding execution unit pipelines, and retire up to four RISC86 operations. The register execution units are shared between six execution pipelines. A maximum of two of these register operations can be issued at a time.

When managing the 24 RISC86 operations, the ICU uses 69 physical registers contained within the RISC86 microarchitecture. Forty-eight of the physical registers are located in a general register file and are grouped as 24 committed or architectural registers plus 24 rename registers. The 24 architectural registers consist of 16 scratch registers and 8 registers that correspond to the x86 general-purpose registers—EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. There is an analogous set of registers specifically for MMX and 3DNow! operations. There are 9 MMX/3DNow! committed or architectural registers plus 12 MMX/3DNow! rename registers. The 9 architectural registers consist of one scratch register and 8 registers that correspond to the MMX registers (mm0–mm7).

The processor offers sophisticated dynamic branch logic that includes the following elements:

■ Branch history/prediction table

■ Branch target cache

■ Return address stack

These components serve to minimize or eliminate the delays due to the branch instructions (jumps, calls, returns) common in x86 software.

The processor implements a two-level branch prediction scheme based on an 8192-entry branch history table. The branch history table stores prediction information that is used for predicting the direction of conditional branches. The target addresses of conditional and unconditional branches are not predicted, but instead are calculated on-the-fly during instruction decode by special branch target address ALUs. The branch target cache augments performance of taken branches by avoiding a one-cycle cache-fetch penalty. This specialized target cache does this by supplying the first 16 bytes of target instructions to the decoders when a branch is taken.

The return address stack serves to optimize CALL and RETURN instruction pairs by remembering the return address of each CALL within a nested series of subroutines and supplying it as the predicted target address of the corresponding RETURN instruction.

As shown in Figure 1 on page 8, the high-performance, out-of-order execution engine is mated to a split 64-Kbyte writeback level-one cache (Harvard architecture) with 32 Kbytes of instruction cache and 32 Kbytes of data cache. The level-one instruction cache feeds the decoders and, in turn, the decoders feed the scheduler. The ICU controls the issue and retirement of RISC86 operations contained in the centralized scheduler buffer. The level-one data cache satisfies most memory reads and writes by the load and store execution units. The store queue temporarily buffers memory writes from the store unit until they can safely be committed into the cache (that is, when all preceding operations have been found to be free of faults and branch mispredictions). The system bus interface is the industry-standard Super7 and Socket 7 interface.

3

AMD-K6®-2 and AMD-K6®-III Processors Execution Units and Dependency Latencies

The AMD-K6-2 and AMD-K6-III processors contain several specialized execution pipelines—store, load, register X, register Y, floating-point, and branch condition. Each pipeline operates independently and handles a specific subset of the RISC86 instruction set. The register X and register Y pipelines each contain integer, multimedia, and 3DNow! technology execution resources, some of which are shared between the two. This chapter describes the operation of these units, their execution latencies, and how these latencies affect concurrent dependency chains.

Note: *meu–Multimedia execution units execute MMX and 3DNow! instructions.*

A dependency occurs when data needed in one execution unit/resource is being processed in another unit/resource (or a different stage of the same unit/resource). Additional latencies can occur because the dependent execution unit must wait for the data from the supplying unit. Table 1 on page 20 provides a summary of the execution units, the operations performed within these units, the operation latency, and the operation throughput.

Execution Unit Terminology

Introduction

The execution units operate with two different types of register values—operands and results. Of these there are three types of operands and two types of results.

Operands

The three types of operands are as follows:

- *Address register operands*—used for address calculations of load and store operations
- *Data register operands*—used for register operations
- *Store data register operands*—used for memory stores

Results

The two types of results are as follows:

- *Data register results*—produced by load or register operations
- *Address register results*—produced by Lea or Push operations

The following examples illustrate the operand and result definitions:

```
Add   AX, BX
```

The Add operation has two data register operands (AX and BX) and one data register result (AX).

```
Load  BX, [SP+4·CX+8]
```

The Load operation has two address register operands (SP and CX as base and index registers, respectively) and a data register result (BX).

```
Store [SP+4·CX+8], AX
```

The Store operation has a store data register operand (AX) and two address register operands (SP and CX as base and index registers, respectively).

```
Lea   SI, [SP+4·CX+8]
```

The Lea operation (a type of store operation) has address register operands (SP and CX as base and index registers, respectively), and an address register result (SI).

Six-Stage Pipeline

To help visualize the operations within the processor, Figure 2 illustrates the effective pipeline stages. This is a simplified illustration in that the processor contains multiple parallel pipelines (starting after common instruction fetch and x86 decode pipe stages), and these pipelines often execute operations out-of-order with respect to each other. This view of the processor execution pipeline illustrates the effect of execution latencies for various types of operations.

For many instructions, the effective pipeline is seven stages. For register operations that do not require execution stage 2, the effective pipeline is six stages.

Instruction Fetch	x86–>RISC86® Decode	RISC86 Op Issue	Operand Fetch	Execution Stage 1	Execution Stage 2*	Commit

Note: *Execution Stage 2 is optional*

Figure 2.   Processor Pipeline

Register Execution Units

The register execution resources are attached to the register X unit execution pipeline and the register Y unit execution pipeline. Each register execution pipeline has dedicated resources that consist of an integer execution unit and a multimedia/ALU execution unit. In addition, both pipelines can use shared execution units for 3DNow! operations and MMX shift and multiply operations. Figure 3 on page 14 shows the details of the register X and Y execution pipelines.
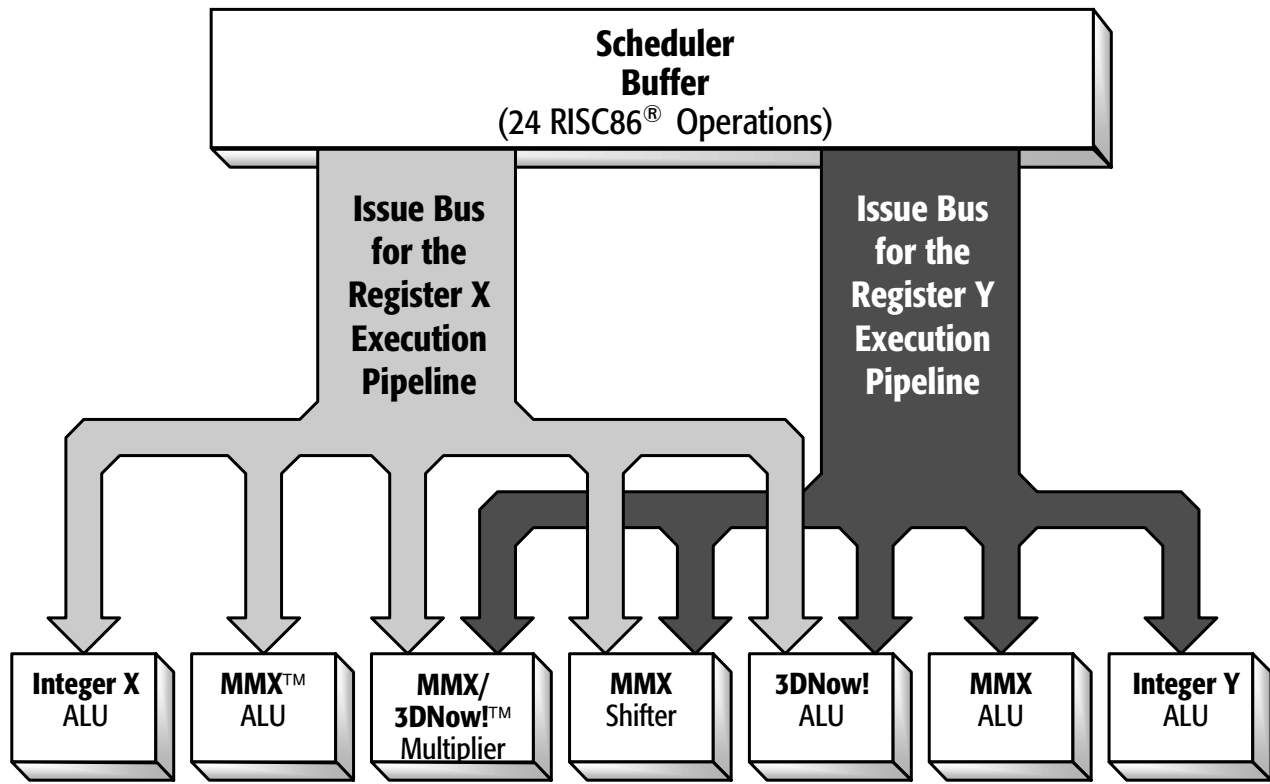
Figure 3.   Register X and Y Functional Units

The register X integer ALU execution resource can execute all ALU operations including ALU, multiply, divide (signed and unsigned), shift, and rotate. Data register results are available after a minimum of one clock of execution latency.

The dedicated integer execution unit contained within the register Y execution pipeline can execute the basic word and doubleword ALU operations (ADD, AND, CMP, OR, SUB, and XOR), zero-extend, and sign-extend operations. Data register results are available after one clock.

The register X and Y execution pipelines each contain a dedicated multimedia execution unit that handles add/subtract, logical, and pack/unpack MMX instructions. The MMX ALU units are symmetrical and can be used simultaneously. This means that the processor can execute two MMX ALU operations each clock cycle.

A number of execution resources are available to both the register X and Y execution pipelines. These shared resources

include the MMX shifter, 3DNow! ALU, and the combined MMX/3DNow! multiplier. Figure 5 on page 16 shows which instruction types are associated with the various execution pipelines.

Any combination of two operations that do not utilize the same shared execution resource can be issued and executed simultaneously. For example, the following pairs of register operations can execute together: MMX logical and 3DNow! add, 3DNow! add and 3DNow! multiply, MMX multiply and 3DNow! add, etc. If issued simultaneously, the following examples result in resource contentions and the stall of one RISC86 operation: MMX multiply and 3DNow! multiply, two MMX multiplies, two 3DNow! multiplies, two 3DNow! adds, etc.

Figure 4 shows the data flow architecture of the single-stage or double-stage integer execution unit pipeline. There are few operations (such as integer multiply) that require a second execution stage. The operation issue and operand fetch stages (execution stage 0) that precede execution stage 1 are not part of the execution pipeline. The data register result is produced near the end of the execution pipe stage.
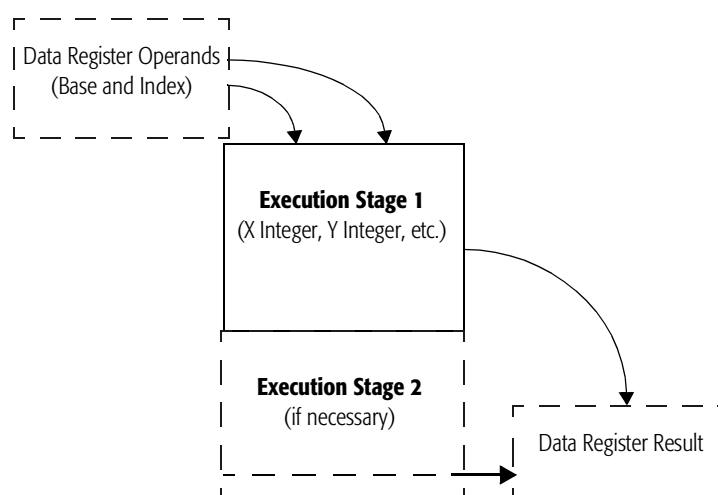
Figure 4.   Register X and Y Execution Stages

Register X Execution Pipeline

Register Y Execution Pipeline

Integer ALU

Integer Shift

Integer Multiply and Divide

Integer Byte Operations

Integer Special Registers

Integer Segment Register Loads

MMX ALU Add/Subtract, Compare

MMX ALU Logical, Pack, Unpack

3DNow!™ Add/Subtract, Compare, Integer Conversion, Reciprocal and Reciprocal Square Root Table Lookup

MMX™ and 3DNow! Multiply, Reciprocal and Reciprocal Square Root Iteration

MMX Shifter

Integer ALU

MMX ALU Add/Subtract, Compare

MMX ALU Logical, Pack, Unpack

Shared Register X and Y Resources

Dedicated Register X Resources

Dedicated Register Y Resources

Figure 5. Microarchitecture and Execution Resources

Load Unit

The load unit is a two-stage pipelined design that performs data memory reads. It has a two-clock latency from the time it receives the address register operands until it produces a data register result on a Dcache hit. A cache miss produces longer latencies. The load unit and the Dcache support hit-under-miss operations where a load operation bypasses a previous load operation that is stalled waiting for a cache line refill. This unit uses two address register operands and a memory data value as inputs, and produces a data register result.

Memory read data can come from either the data cache or from the store queue entry (for a recent store). If the data is forwarded from the store queue, there is zero additional execution latency, which means that a dependent load operation can complete its execution one clock after a store operation completes execution.

Figure 6 shows the architecture of the two-stage load execution pipeline. The address register operands are received at the end of the operand fetch pipe stage, and the data register result is produced near the end of the second execution pipe stage. The operation issue and fetch stages that precede these execution stages are not shown.
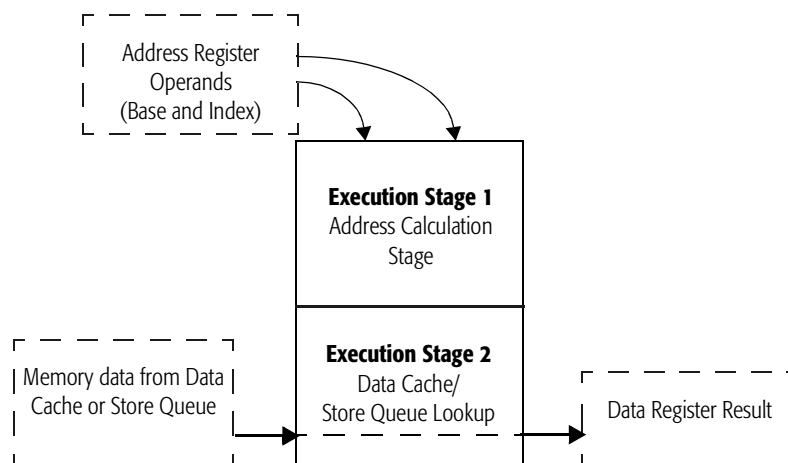
Figure 6. Load Execution Unit

Store Unit

The store execution unit is a two-stage pipelined design that performs data memory writes and, in some cases, produces an address register result. For inputs, the store unit uses two address register operands and, during actual memory writes, a store data register operand. This unit also produces an address register result for some store unit operations. For most store operations, for example those that write data to memory, the store unit produces a physical memory address and the associated data bytes to be written. After execution completes, these results are entered in a new store queue entry. The store queue can hold up to seven data results, each of which can be 64 bits.

The store unit has a one-clock execution latency from the time it receives address register operands until the time it produces an address register result. The most common examples are the Load Effective Address (Lea) and Store and Update (Push) RISC86 operations, which are produced from the x86 LEA and PUSH instructions, respectively. Most store operations do not produce an address register result and only perform a memory write. The Push operation is unique because it produces an address register result and performs a memory write.

The store unit has a one-clock execution latency from the time it receives a store data operand until it enters the store memory address and data pair into the store queue.

The store unit can have a three-clock latency from the time it receives address register operands and a store data register operand until it enters the store memory address and data pair into the store queue.

Note: Address register operands are required at the start of execution, but register store data is not required until the end of execution.

Figure 7 on page 19 shows the architecture of the two-stage store execution pipeline. The operation issue and fetch stages that precede this execution stage are not part of the execution pipeline. The address register operands are received at the end of the operand fetch pipe stage, and the new store queue entry is created upon completion of the second execution pipe stage.
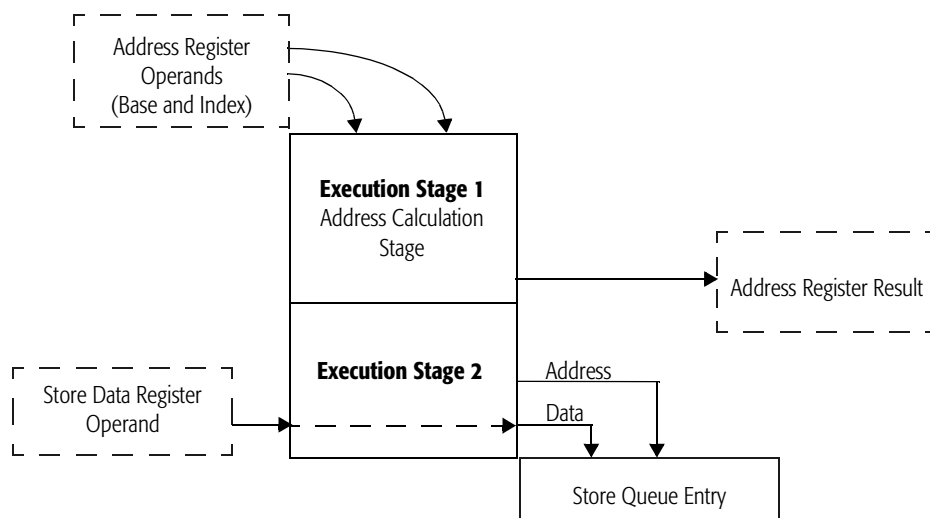
Figure 7. Store Unit Execution Pipeline

Branch Condition Unit

The branch condition unit is separate from the branch prediction logic, which is utilized at x86 instruction decode time. This unit resolves conditional branches, such as JCC and LOOP instructions, at a rate of up to one per clock cycle. This unit has a dedicated RISC86 issue bus from the scheduler.

Floating-Point Unit

The floating-point unit (FPU) handles all register operations for x87 instructions. The execution unit is a single-stage design that takes data register operands as inputs and produces a data register result as an output. The most common floating-point instructions have a two clock execution latency from the time the FPU receives data register operands until it produces a data register result. The FPU has its own RISC86 issue bus from the scheduler.

Latencies and Throughput

Table 1 summarizes the static latencies and throughput of each execution unit.

Table 1.   RISC86® Execution Latencies and Throughput

Execution Unit	Operations	Latency	Throughput
Register X Integer Unit	Integer ALU	1	1
	Integer Multiply	2–3	2–3
	Integer Shift	1	1
Register X Multimedia Unit	MMX™ Add/Subtract	1	1
	MMX Logical, Pack, Unpack	1	1
Register Y Integer Unit	Integer ALU (16– and 32– bit operands)	1	1
Register Y Multimedia Unit	MMX Add/Subtract	1	1
	MMX Logical, Pack, Unpack	1	1
Multimedia/3DNow!™ Shared Execution Units (X and Y)	MMX Shifter	1	1
	MMX/3DNow! Multiply, Reciprocal and, Reciprocal Square Root Iteration	2	1
	3DNow! Add, Compare, Integer Conversion, Reciprocal, and Reciprocal Square Root Table Lookup	2	1
Load	From Address Register Operands to Data Register Result	2	1
	Memory Read Data from Data Cache/Store Queue to Data Register Result	0	1
Store	From Address Register Operands to Address Register Result	1	1
	From Store Data Register Operand to Store Queue Entry	1	1
	From Address Register Operands to Store Queue Entry	3	1
Branch	Resolves Branch Conditions	1	1
FPU	FADD, FSUB	2	2
	FMUL	2	2

Note:

No additional latency exists between execution of dependent operations. Bypassing of register results directly from producing execution units to the operand inputs of dependent units is fully supported. Similarly, forwarding of memory store values from the store queue to dependent load operations is supported.

Resource Constraints

To optimize code effectively, execution resource constraints must be considered. Due to a fixed number of execution units, even with up to six RISC86 operations per cycle, optimal execution parallelism should be carefully scheduled.

For example, if an IMUL is decoded and issued to the X pipeline, for the next two to three cycles integer, MMX, and 3DNow! technology RISC86 operations can only be issued to the Y pipeline. Another example is two ALU instructions that require the load unit. Only one load can occur each cycle, therefore, one instruction would stall for a cycle.

Contention for execution resources can cause delays in the issuing and execution of instructions. In addition, stalls due to resource constraints can increase dependency latencies to cause or exacerbate stalls due to dependencies. In general, constraints that delay non-critical instructions do not impact performance because such stalls typically overlap with the execution of critical operations.

Code Sample Analysis

The samples in this section show the execution behavior of several series of instructions as a function of decode constraints, dependencies, and execution resource constraints.

The sample tables show the x86 instructions, the RISC86 operation equivalents and a description of the events occurring within the processor.

The following nomenclature is used to describe the current location of a RISC86 operation:

- D — Decode stage
- I_X — Issue stage of register X unit
- O_X — Operand fetch stage of register X unit
- E_{X1} — Execution stage 1 of register X unit
- E_{X2} — Execution stage 2 of register X unit
- I_Y — Issue stage of register Y unit
- O_Y — Operand fetch stage of register Y unit
- E_{Y1} — Execution stage 1 of register Y unit
- E_{Y2} — Execution stage 2 of register Y unit
- I_L — Issue stage of load unit
- O_L — Operand fetch stage of load unit
- E_{L1} — Execution stage 1 of load unit
- E_{L2} — Execution stage 2 of load unit
- I_S — Issue stage of store unit
- O_S — Operand fetch stage of store unit
- E_{S1} — Execution stage 1 of store unit
- E_{S2} — Execution stage 2 of store unit

Note: *Instructions execute more efficiently (that is, without delays) when scheduled apart by suitable distances based on dependencies. In general, the samples in this section show poorly scheduled code in order to illustrate the resultant effects.*

Table 2. Sample 1 – Integer Register Operations

Instruction Number	Instruction	RISC86® Operation	Clocks								
			1	2	3	4	5	6	7	8	9
1	IMUL EAX, EBX	alux	D	D	I_X	O_X	E_{X1}				
		alux				I_X	O_X	E_{X1}			
		alux					I_X	O_X	E_{X1}		
2	INC ESI	alu			D	I_Y	O_Y	E_{Y1}			
3	MOV EDI, 0x07F4	limm			D						
4	SHL EAX, 8	alux				D		I_X	O_X	E_{X1}	
5	OR EAX, 0x0F	alu				D	I_Y	O_Y	I_X	O_X	E_{X1}
6	ADD ESI, EDX	alu					D	I_Y	O_Y	E_{Y1}	
7	SUB EDI, ECX	alu					D		I_Y	O_Y	E_{Y1}

Comments for Each Instruction Number

1   It takes two decode cycles because IMUL is vector decoded. The IMUL instruction is executable only in the integer X unit. It is a non-pipelined 2–3 cycle latency register operation that is equivalent to three serially-dependent register operations (the result of the second and third operations are EAX and EDX, respectively).

2   This simple alu operation ends up in the Y pipe.

3   A load immediate (limm) RISC86 operation does not require execution. The result value is immediately available to dependent operations.

4   Shift instructions are only executable in the integer X unit. Issue is delayed by preceding IMUL operations due to a resource constraint of the integer X unit.

5   The register operation is bumped out of the integer Y unit in clock 6 because it must wait for more than one cycle for its dependencies to resolve. It is reissued in the next cycle to the integer X unit (just in time for availability of its operand).

6   This add alu falls through to the integer Y unit right behind the first issuance of instruction #5 without delay (as a result of instruction #5 being bumped out of the way).

7   The issuance of the subtract register operation is delayed in clock 6 due to the resource constraints of the integer Y unit.

Table 3. Sample 2 – Integer Register and Memory Load Operations

Instruction Number	Instruction		RISC86® Operation	Clocks										
				1	2	3	4	5	6	7	8	9	10	11
1	DEC	EDX	alu	D	I_X	O_X	E_{X1}							
2	MOV	EDI, [ECX]	load	D	I_L	O_L	E_{L1}	E_{L2}						
3	SUB	EAX, [EDX+20]	load		D	I_L	O_L	E_{L1}	E_{L2}					
			alu			I_X	O_X	I_X	O_X	E_{X1}				
4	SAR	EAX, 5	alux		D		I_X	O_X	I_X	O_X	E_{X1}			
5	ADD	ECX, [EDI+4]	load			D	I_L	O_L	E_{L1}	E_{L2}				
			alu				I_Y	O_Y	I_Y	O_Y	E_{Y1}			
6	AND	EBX, 0x1F	alu			D		I_Y	O_Y	E_{Y1}				
7	MOV	ESI, [0x0F100]	load				D	I_L	O_L	E_{L1}	E_{L2}			
8	OR	ECX, [ESI+EAX*4+8]	load					D		I_L	O_L	O_L	E_{L1}	E_{L2}
			alu							I_X	O_X	I_X	O_X	E_{X1}

Comments for Each Instruction Number

1   This simple alu operation ends up in the X pipe.

2   This operation occupies the load execution unit.

3   The register operand for the load operation is bypassed, without delay, from the result of instruction #1's register operand. In clock 4, the register operation is bumped out of the integer X unit while waiting for the previous load operation result to complete. It is reissued just in time to receive the bypassed result of the load.

4   Shift instructions are only executable in the integer X unit. The register operation is bumped in clock 5 while waiting for the result of the preceding instruction #3.

5   The register operand for the load operation is bypassed, without delay, from the result of instruction #2's register operand. This and most surrounding load operations are generated by instruction decoders, and issued and smoothly executed by the load unit at a rate of one clock per cycle. In clock 5, the register operation is bumped out of the integer Y unit while waiting for the previous load operation result to complete.

6   The register operation falls through into the integer Y unit right behind instruction #5's register operation.

7   This operation falls into the load unit behind the load in instruction #5.

8   The operand fetch for the load operation is delayed because it needs the result of the immediately preceding load operation #7 as well as the results from earlier instruction #4.

Table 4. Sample 3 – Integer Register and Memory Load/Store Operations

Instruction Number	Instruction	RISC86® Operation	Clocks										
			1	2	3	4	5	6	7	8	9	10	11
1	MOV EDX, [0xA0008F00]	load	D	I_L	O_L	E_{L1}	E_{L2}						
2	ADD [EDX+16], 7	load		D	I_L	O_L	O_L	E_{L1}	E_{L2}				
		alu			I_X	O_X	I_X	O_X	O_X	E_{X1}			
		store			I_S	O_S	O_S	E_{S1}	E_{S2}	E_{S2}			
3	SUB EAX, [EDX+16]	load			D	I_L	I_L	O_L	E_{L1}	E_{L2}	E_{L2}		
		alu				I_X	O_X	I_X	I_X	O_X	O_X	E_{X1}	
4	PUSH EAX	store			D	I_S	I_S	O_S	E_{S1}	E_{S2}	E_{S2}	E_{S2}	
5	LEA EBX, [ECX+EAX*4+3]	store				D		I_S	O_S	O_S	O_S	E_{S1}	E_{S2}
6	MOV EDI, EBX	alu				D	I_Y	O_Y	I_Y	O_Y	I_X	O_X	E_{X1}

Comments for Each Instruction Number

1   This operation occupies the load unit.

2   This long-decoded ADD instruction takes a single clock to decode. The operand fetch for the load operation is delayed waiting for the result of the previous load operation from instruction #1. The store operation completes concurrent with the register operation. The result of the register operation is bypassed directly into a new store queue entry created by the store operation.

3   The issue of the load operation is delayed because the operand fetch of the preceding load operation from instruction #2 was delayed. The completion of the load operation is held up due to a memory dependency on the preceding store operation of instruction #2. The load operation completes immediately after the store operation, with the store data being forwarded from a new store queue entry.

4   Completion of the store operation is held up due to a data dependency on the preceding instruction #3. The store data is bypassed directly into a new store queue entry from the result of instruction #3's register operation.

5   The Lea RISC86 operation is executed by the store unit. The operand fetch is delayed waiting for the result of instruction #3. The register result value is produced in the first execution stage of the store unit.

6   This simple alu operation is stalled due to the dependency of the EBX result in instruction #5.

Table 5. Sample 4 – Integer, MMX™, and Memory Load/Store Operations

Inst. Num.	Instruction	RISC86® Operation	Clocks											
			1	2	3	4	5	6	7	8	9	10	11	12
1	PADDSW MM0, MM4	alu	D	I_X	O_X	E_{X1}								
2	PADDSW MM1, MM5	alu	D	I_Y	O_Y	E_{Y1}								
3	PSRAW MM0, 3	alu		D	I_X	O_X	E_{X1}							
4	MOVQ MM2, [EAX+EBX]	mload		D	I_L	O_L	E_{L1}	E_{L2}						
5	PAND MM0, MM3	alu			D	I_X	O_X	E_{X1}						
6	PMULLW MM2, [EDI+8]	mload			D	I_L	O_L	E_{L1}	E_{L2}					
		alu				I_Y	O_Y	I_X	O_X	E_{X1}	E_{X2}			
7	MOVQ [ESP+4], MM2	mstore				D	I_S	O_S	E_{S1}	E_{S2}	E_{S2}			
8	ADD EBX, ECX	alu				D	I_X	O_X	E_{X1}					
9	PMULLW MM6, MM7	alu					D	I_Y	O_Y	E_{Y1}	E_{Y1}	E_{Y2}		
10	PMADDWD MM2, MM6	alu					D		I_X	O_X	O_X	O_X	E_{X1}	E_{X2}

Comments for Each Instruction Number

1, 2   Instructions 1 and 2 are decoded, issued, and executed simultaneously and in parallel due to no decode restrictions, dependency delays, or execution resource constraints.

3   This instruction is decoded, issued, and executed without delay, one cycle behind the preceding one-cycle execution latency instruction on which it is dependent.

4   This multimedia operation occupies the load unit.

5   This instruction is decoded, issued, and executed without delay, right behind the preceding operations on which it is dependent.

6   This and the preceding instruction are decoded and issued together without delay. The operand fetch of the register operation is delayed because of the dependency on the associated load. As a result, the register operation is bumped out of register unit Y in clock 5 and is reissued in the next cycle to register unit X (as it happens), just in time for availability of its operands.

7   Completion of this store operation is held up due to a data dependency on the preceding MMX multiply register operation (which has a two-cycle execution latency). The store data is bypassed directly into a new store queue entry from the result of the register operation.

8   This operation is issued to register unit X and executes without delay and out-of-order with respect to the preceding register operation from instruction #6 (which was bumped out of the way while waiting for its operands).

9   This MMX multiply register operation issues to and starts execution in register unit Y in parallel with an MMX multiply register operation from instruction #6 which simultaneously issues to and starts execution in register unit X. Due to an execution resource constraint, this operation is delayed one cycle in its first execution pipe stage and then executes and completes normally, one cycle behind the other contending register operation. (This takes advantage of the pipelined nature of the MMX multiply execution logic.)

10   The issue of this operation is delayed (in clock 6) for one cycle due to two earlier register operations being selected for issue. It is then delayed further during operand fetch while waiting for the preceding two-cycle latency MMX multiply register operations to complete execution.

4

Instruction Dispatch

This chapter describes the RISC86 operations executed by each instruction. Tables 6 through 9 starting on page 29 define the integer, MMX, floating-point, and 3DNow! instructions. Only the AMD-K6-2 and AMD-K6-III processors support the instructions in Table 9, "3DNow!™ Instructions," on page 55.

The first column in these tables indicates the instruction mnemonic and operand types with the following notations:

■ *reg8*—byte integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte

■ *mreg8*—byte integer register or byte integer value in memory defined by the modR/M byte

■ *reg16/32*—word or doubleword integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte

■ *mreg16/32*—word or doubleword integer register, or word or doubleword integer value in memory defined by the modR/M byte

■ *mem8*—byte integer value in memory

■ *mem16/32*—word or doubleword integer value in memory

■ *mem32/48*—doubleword or 48-bit integer value in memory

■ *mem48*—48-bit integer value in memory

■ *mem64*—64-bit value in memory

■ *imm8*—8-bit immediate value

■ *imm16/32*—16-bit or 32-bit immediate value

- *disp8*—8-bit displacement value
- *disp16/32*—16-bit or 32-bit displacement value
- *disp32/48*—doubleword or 48-bit displacement value
- *eXX*—register width depending on the operand size
- *mem32real*—32-bit floating-point value in memory
- *mem64real*—64-bit floating-point value in memory
- *mem80real*—80-bit floating-point value in memory
- *mmreg*—MMX/3DNow! register
- *mmreg1*—MMX/3DNow! register defined by bits 5, 4, and 3 of the modR/M byte
- *mmreg2*—MMX/3DNow! register defined by bits 2, 1, and 0 of the modR/M byte

The second and third columns list all applicable encoding opcode bytes.

The fourth column lists the modR/M byte when used by the instruction. The modR/M byte defines the instruction as register or memory form. If mod bits 7 and 6 are documented as mm (memory form), mm can only be 10b, 01b, or 00b.

The fifth column lists the type of instruction decode—short, long, or vector. The processor decode logic can process two short, one long, or one vector decode per clock. Any pair of short decodable instructions, be it integer, floating-point, MMX, or 3DNow!, can be decoded simultaneously. All MMX and 3DNow! instructions are short decodable except the EMMS, FEMMS, and PREFETCH instructions.

The sixth column lists the type of RISC86 operation(s) required for the instruction. The operation types and corresponding execution units are as follows:

- *load, fload, mload*—load unit
- *store, fstore, mstore*—store unit
- *alu*—either of the integer register execution units
- *alux*—integer register X execution unit only
- *branch*—branch condition unit
- *float*—floating-point execution unit
- *meu*—Multimedia execution units for MMX and 3DNow! instructions
- *limm*—load immediate, instruction control unit only

The operation(s) of most instructions form a single dependency chain. For instructions whose operations form two parallel dependency chains, the RISC86 operations for each dependency chain is shown on a separate row.

Table 6.   Integer Instructions

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
AAA	37h			vector	
AAD	D5h	0Ah		vector	
AAM	D4h	0Ah		vector	
AAS	3Fh			vector	
ADC mreg8, reg8	10h		11-xxx-xxx	vector	
ADC mem8, reg8	10h		mm-xxx-xxx	vector	
ADC mreg16/32, reg16/32	11h		11-xxx-xxx	vector	
ADC mem16/32, reg16/32	11h		mm-xxx-xxx	vector	
ADC reg8, mreg8	12h		11-xxx-xxx	vector	
ADC reg8, mem8	12h		mm-xxx-xxx	vector	
ADC reg16/32, mreg16/32	13h		11-xxx-xxx	vector	
ADC reg16/32, mem16/32	13h		mm-xxx-xxx	vector	
ADC AL, imm8	14h			vector	
ADC EAX, imm16/32	15h			vector	
ADC mreg8, imm8	80h		11-010-xxx	vector	
ADC mem8, imm8	80h		mm-010-xxx	vector	
ADC mreg16/32, imm16/32	81h		11-010-xxx	vector	
ADC mem16/32, imm16/32	81h		mm-010-xxx	vector	
ADC mreg16/32, imm8 (signed ext.)	83h		11-010-xxx	vector	
ADC mem16/32, imm8 (signed ext.)	83h		mm-010-xxx	vector	
ADD mreg8, reg8	00h		11-xxx-xxx	short	alux
ADD mem8, reg8	00h		mm-xxx-xxx	long	load, alux, store
ADD mreg16/32, reg16/32	01h		11-xxx-xxx	short	alu
ADD mem16/32, reg16/32	01h		mm-xxx-xxx	long	load, alu, store
ADD reg8, mreg8	02h		11-xxx-xxx	short	alux
ADD reg8, mem8	02h		mm-xxx-xxx	short	load, alux
ADD reg16/32, mreg16/32	03h		11-xxx-xxx	short	alu
ADD reg16/32, mem16/32	03h		mm-xxx-xxx	short	load, alu
ADD AL, imm8	04h			short	alux

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
ADD EAX, imm16/32	05h			short	alu
ADD mreg8, imm8	80h		11-000-xxx	short	alux
ADD mem8, imm8	80h		mm-000-xxx	long	load, alux, store
ADD mreg16/32, imm16/32	81h		11-000-xxx	short	alu
ADD mem16/32, imm16/32	81h		mm-000-xxx	long	load, alu, store
ADD mreg16/32, imm8 (signed ext.)	83h		11-000-xxx	short	alux
ADD mem16/32, imm8 (signed ext.)	83h		mm-000-xxx	long	load, alux, store
AND mreg8, reg8	20h		11-xxx-xxx	short	alux
AND mem8, reg8	20h		mm-xxx-xxx	long	load, alux, store
AND mreg16/32, reg16/32	21h		11-xxx-xxx	short	alu
AND mem16/32, reg16/32	21h		mm-xxx-xxx	long	load, alu, store
AND reg8, mreg8	22h		11-xxx-xxx	short	alux
AND reg8, mem8	22h		mm-xxx-xxx	short	load, alux
AND reg16/32, mreg16/32	23h		11-xxx-xxx	short	alu
AND reg16/32, mem16/32	23h		mm-xxx-xxx	short	load, alu
AND AL, imm8	24h			short	alux
AND EAX, imm16/32	25h			short	alu
AND mreg8, imm8	80h		11-100-xxx	short	alux
AND mem8, imm8	80h		mm-100-xxx	long	load, alux, store
AND mreg16/32, imm16/32	81h		11-100-xxx	short	alu
AND mem16/32, imm16/32	81h		mm-100-xxx	long	load, alu, store
AND mreg16/32, imm8 (signed ext.)	83h		11-100-xxx	short	alux
AND mem16/32, imm8 (signed ext.)	83h		mm-100-xxx	long	load, alux, store
ARPL mreg16, reg16	63h		11-xxx-xxx	vector	
ARPL mem16, reg16	63h		mm-xxx-xxx	vector	
BOUND	62h			vector	
BSF reg16/32, mreg16/32	0Fh	BCh	11-xxx-xxx	vector	
BSF reg16/32, mem16/32	0Fh	BCh	mm-xxx-xxx	vector	
BSR reg16/32, mreg16/32	0Fh	BDh	11-xxx-xxx	vector	
BSR reg16/32, mem16/32	0Fh	BDh	mm-xxx-xxx	vector	
BSWAP EAX	0Fh	C8h		long	alu
BSWAP ECX	0Fh	C9h		long	alu
BSWAP EDX	0Fh	CAh		long	alu

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
BSWAP EBX	0Fh	CBh		long	alu
BSWAP ESP	0Fh	CCh		long	alu
BSWAP EBP	0Fh	CDh		long	alu
BSWAP ESI	0Fh	CEh		long	alu
BSWAP EDI	0Fh	CFh		long	alu
BT mreg16/32, reg16/32	0Fh	A3h	11-xxx-xxx	vector	
BT mem16/32, reg16/32	0Fh	A3h	mm-xxx-xxx	vector	
BT mreg16/32, imm8	0Fh	BAh	11-100-xxx	vector	
BT mem16/32, imm8	0Fh	BAh	mm-100-xxx	vector	
BTC mreg16/32, reg16/32	0Fh	BBh	11-xxx-xxx	vector	
BTC mem16/32, reg16/32	0Fh	BBh	mm-xxx-xxx	vector	
BTC mreg16/32, imm8	0Fh	BAh	11-111-xxx	vector	
BTC mem16/32, imm8	0Fh	BAh	mm-111-xxx	vector	
BTR mreg16/32, reg16/32	0Fh	B3h	11-xxx-xxx	vector	
BTR mem16/32, reg16/32	0Fh	B3h	mm-xxx-xxx	vector	
BTR mreg16/32, imm8	0Fh	BAh	11-110-xxx	vector	
BTR mem16/32, imm8	0Fh	BAh	mm-110-xxx	vector	
BTS mreg16/32, reg16/32	0Fh	ABh	11-xxx-xxx	vector	
BTS mem16/32, reg16/32	0Fh	ABh	mm-xxx-xxx	vector	
BTS mreg16/32, imm8	0Fh	BAh	11-101-xxx	vector	
BTS mem16/32, imm8	0Fh	BAh	mm-101-xxx	vector	
CALL full pointer	9Ah			vector	
CALL near imm16/32	E8h			short	store
CALL mem16:16/32	FFh		11-011-xxx	vector	
CALL near mreg32 (indirect)	FFh		11-010-xxx	vector	
CALL near mem32 (indirect)	FFh		mm-010-xxx	vector	
CBW/CWDE EAX	98h			vector	
CLC	F8h			vector	
CLD	FCh			vector	
CLI	FAh			vector	
CLTS	0Fh	06h		vector	
CMC	F5h			vector	
CMP mreg8, reg8	38h		11-xxx-xxx	short	alux

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
CMP mem8, reg8	38h		mm-xxx-xxx	short	load, alux
CMP mreg16/32, reg16/32	39h		11-xxx-xxx	short	alu
CMP mem16/32, reg16/32	39h		mm-xxx-xxx	short	load, alu
CMP reg8, mreg8	3Ah		11-xxx-xxx	short	alux
CMP reg8, mem8	3Ah		mm-xxx-xxx	short	load, alux
CMP reg16/32, mreg16/32	3Bh		11-xxx-xxx	short	alu
CMP reg16/32, mem16/32	3Bh		mm-xxx-xxx	short	load, alu
CMP AL, imm8	3Ch			short	alux
CMP EAX, imm16/32	3Dh			short	alu
CMP mreg8, imm8	80h		11-111-xxx	short	alux
CMP mem8, imm8	80h		mm-111-xxx	short	load, alux
CMP mreg16/32, imm16/32	81h		11-111-xxx	short	alu
CMP mem16/32, imm16/32	81h		mm-111-xxx	short	load, alu
CMP mreg16/32, imm8 (signed ext.)	83h		11-111-xxx	long	load, alu
CMP mem16/32, imm8 (signed ext.)	83h		mm-111-xxx	long	load, alu
CMPSB mem8,mem8	A6h			vector	
CMPSW mem16, mem32	A7h			vector	
CMPSD mem32, mem32	A7h			vector	
CMPXCHG mreg8, reg8	0Fh	B0h	11-xxx-xxx	vector	
CMPXCHG mem8, reg8	0Fh	B0h	mm-xxx-xxx	vector	
CMPXCHG mreg16/32, reg16/32	0Fh	B1h	11-xxx-xxx	vector	
CMPXCHG mem16/32, reg16/32	0Fh	B1h	mm-xxx-xxx	vector	
CMPXCHG8B EDX:EAX	0Fh	C7h	11-xxx-xxx	vector	
CMPXCHG8B mem64	0Fh	C7h	mm-xxx-xxx	vector	
CPUID	0Fh	A2h		vector	
CWD/CDQ EDX, EAX	99h			vector	
DAA	27h			vector	
DAS	2Fh			vector	
DEC EAX	48h			short	alu
DEC ECX	49h			short	alu
DEC EDX	4Ah			short	alu
DEC EBX	4Bh			short	alu
DEC ESP	4Ch			short	alu

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
DEC EBP	4Dh			short	alu
DEC ESI	4Eh			short	alu
DEC EDI	4Fh			short	alu
DEC mreg8	FEh		11-001-xxx	vector	
DEC mem8	FEh		mm-001-xxx	long	load, alux, store
DEC mreg16/32	FFh		11-001-xxx	vector	
DEC mem16/32	FFh		mm-001-xxx	long	load, alu, store
DIV AL, mreg8	F6h		11-110-xxx	vector	
DIV AL, mem8	F6h		mm-110-xxx	vector	
DIV EAX, mreg16/32	F7h		11-110-xxx	vector	
DIV EAX, mem16/32	F7h		mm-110-xxx	vector	
IDIV mreg8	F6h		11-111-xxx	vector	
IDIV mem8	F6h		mm-111-xxx	vector	
IDIV EAX, mreg16/32	F7h		11-111-xxx	vector	
IDIV EAX, mem16/32	F7h		mm-111-xxx	vector	
IMUL reg16/32, imm16/32	69h		11-xxx-xxx	vector	
IMUL reg16/32, mreg16/32, imm16/32	69h		11-xxx-xxx	vector	
IMUL reg16/32, mem16/32, imm16/32	69h		mm-xxx-xxx	vector	
IMUL reg16/32, imm8 (sign extended)	6Bh		11-xxx-xxx	vector	
IMUL reg16/32, mreg16/32, imm8 (signed)	6Bh		11-xxx-xxx	vector	
IMUL reg16/32, mem16/32, imm8 (signed)	6Bh		mm-xxx-xxx	vector	
IMUL AX, AL, mreg8	F6h		11-101-xxx	vector	
IMUL AX, AL, mem8	F6h		mm-101-xxx	vector	
IMUL EDX:EAX, EAX, mreg16/32	F7h		11-101-xxx	vector	
IMUL EDX:EAX, EAX, mem16/32	F7h		mm-101-xxx	vector	
IMUL reg16/32, mreg16/32	0Fh	AFh	11-xxx-xxx	vector	
IMUL reg16/32, mem16/32	0Fh	AFh	mm-xxx-xxx	vector	
IN AL, imm8	E4h			vector	
IN AX, imm8	E5h			vector	
IN EAX, imm8	E5h			vector	
IN AL, DX	ECh			vector	
IN AX, DX	EDh			vector	
IN EAX, DX	EDh			vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
INC EAX	40h			short	alu
INC ECX	41h			short	alu
INC EDX	42h			short	alu
INC EBX	43h			short	alu
INC ESP	44h			short	alu
INC EBP	45h			short	alu
INC ESI	46h			short	alu
INC EDI	47h			short	alu
INC mreg8	FEh		11-000-xxx	vector	
INC mem8	FEh		mm-000-xxx	long	load, alux, store
INC mreg16/32	FFh		11-000-xxx	vector	
INC mem16/32	FFh		mm-000-xxx	long	load, alu, store
INVD	0Fh	08h		vector	
INVLPG	0Fh	01h	mm-111-xxx	vector	
JO short disp8	70h			short	branch
JB/JNAE short disp8	71h			short	branch
JNO short disp8	71h			short	branch
JNB/JAE short disp8	73h			short	branch
JZ/JE short disp8	74h			short	branch
JNZ/JNE short disp8	75h			short	branch
JBE/JNA short disp8	76h			short	branch
JNBE/JA short disp8	77h			short	branch
JS short disp8	78h			short	branch
JNS short disp8	79h			short	branch
JP/JPE short disp8	7Ah			short	branch
JNP/JPO short disp8	7Bh			short	branch
JL/JNGE short disp8	7Ch			short	branch
JNL/JGE short disp8	7Dh			short	branch
JLE/JNG short disp8	7Eh			short	branch
JNLE/JG short disp8	7Fh			short	branch
JCXZ/JEC short disp8	E3h			vector	
JO near disp16/32	0Fh	80h		short	branch
JNO near disp16/32	0Fh	81h		short	branch

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
JB/JNAE near disp16/32	0Fh	82h		short	branch
JNB/JAE near disp16/32	0Fh	83h		short	branch
JZ/JE near disp16/32	0Fh	84h		short	branch
JNZ/JNE near disp16/32	0Fh	85h		short	branch
JBE/JNA near disp16/32	0Fh	86h		short	branch
JNBE/JA near disp16/32	0Fh	87h		short	branch
JS near disp16/32	0Fh	88h		short	branch
JNS near disp16/32	0Fh	89h		short	branch
JP/JPE near disp16/32	0Fh	8Ah		short	branch
JNP/JPO near disp16/32	0Fh	8Bh		short	branch
JL/JNGE near disp16/32	0Fh	8Ch		short	branch
JNL/JGE near disp16/32	0Fh	8Dh		short	branch
JLE/JNG near disp16/32	0Fh	8Eh		short	branch
JNLE/JG near disp16/32	0Fh	8Fh		short	branch
JMP near disp16/32 (direct)	E9h			short	branch
JMP far disp32/48 (direct)	EAh			vector	
JMP disp8 (short)	EBh			short	branch
JMP far mreg32 (indirect)	EFh		11-101-xxx	vector	
JMP far mem32 (indirect)	EFh		mm-101-xxx	vector	
JMP near mreg16/32 (indirect)	FFh		11-100-xxx	vector	
JMP near mem16/32 (indirect)	FFh		mm-100-xxx	vector	
LAHF	9Fh			vector	
LAR reg16/32, mreg16/32	0Fh	02h	11-xxx-xxx	vector	
LAR reg16/32, mem16/32	0Fh	02h	mm-xxx-xxx	vector	
LDS reg16/32, mem32/48	C5h		mm-xxx-xxx	vector	
LEA reg16/32, mem16/32	8Dh		mm-xxx-xxx	short	load, alu
LEAVE	C9h			long	load, alu, alu
LES reg16/32, mem32/48	C4h		mm-xxx-xxx	vector	
LFS reg16/32, mem32/48	0Fh	B4h		vector	
LGDT mem48	0Fh	01h	mm-010-xxx	vector	
LGS reg16/32, mem32/48	0Fh	B5h		vector	
LIDT mem48	0Fh	01h	mm-011-xxx	vector	
LLDT mreg16	0Fh	00h	11-010-xxx	vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
LLDT mem16	0Fh	00h	mm-010-xxx	vector	
LMSW mreg16	0Fh	01h	11-100-xxx	vector	
LMSW mem16	0Fh	01h	mm-100-xxx	vector	
LODSB AL, mem8	ACh			long	load, alux
LODSW AX, mem16	ADh			long	load, alu
LODSD EAX, mem32	ADh			long	load, alu
LOOP disp8	E2h			short	alu, branch
LOOPE/LOOPZ disp8	E1h			vector	
LOOPNE/LOOPNZ disp8	E0h			vector	
LSL reg16/32, mreg16/32	0Fh	03h	11-xxx-xxx	vector	
LSL reg16/32, mem16/32	0Fh	03h	mm-xxx-xxx	vector	
LSS reg16/32, mem32/48	0Fh	B2h	mm-xxx-xxx	vector	
LTR mreg16	0Fh	00h	11-011-xxx	vector	
LTR mem16	0Fh	00h	mm-011-xxx	vector	
MOV mreg8, reg8	88h		11-xxx-xxx	short	alux
MOV mem8, reg8	88h		mm-xxx-xxx	short	store
MOV mreg16/32, reg16/32	89h		11-xxx-xxx	short	alu
MOV mem16/32, reg16/32	89h		mm-xxx-xxx	short	store
MOV reg8, mreg8	8Ah		11-xxx-xxx	short	alux
MOV reg8, mem8	8Ah		mm-xxx-xxx	short	load
MOV reg16/32, mreg16/32	8Bh		11-xxx-xxx	short	alu
MOV reg16/32, mem16/32	8Bh		mm-xxx-xxx	short	load
MOV mreg16, segment reg	8Ch		11-xxx-xxx	long	load
MOV mem16, segment reg	8Ch		mm-xxx-xxx	vector	
MOV segment reg, mreg16	8Eh		11-xxx-xxx	vector	
MOV segment reg, mem16	8Eh		mm-xxx-xxx	vector	
MOV AL, mem8	A0h			short	load
MOV EAX, mem16/32	A1h			short	load
MOV mem8, AL	A2h			short	store
MOV mem16/32, EAX	A3h			short	store
MOV AL, imm8	B0h			short	limm
MOV CL, imm8	B1h			short	limm
MOV DL, imm8	B2h			short	limm

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
MOV BL, imm8	B3h			short	limm
MOV AH, imm8	B4h			short	limm
MOV CH, imm8	B5h			short	limm
MOV DH, imm8	B6h			short	limm
MOV BH, imm8	B7h			short	limm
MOV EAX, imm16/32	B8h			short	limm
MOV ECX, imm16/32	B9h			short	limm
MOV EDX, imm16/32	BAh			short	limm
MOV EBX, imm16/32	BBh			short	limm
MOV ESP, imm16/32	BCh			short	limm
MOV EBP, imm16/32	BDh			short	limm
MOV ESI, imm16/32	BEh			short	limm
MOV EDI, imm16/32	BFh			short	limm
MOV mreg8, imm8	C6h		11-000-xxx	short	limm
MOV mem8, imm8	C6h		mm-000-xxx	long	store
MOV reg16/32, imm16/32	C7h		11-000-xxx	short	limm
MOV mem16/32, imm16/32	C7h		mm-000-xxx	long	store
MOVSB mem8,mem8	A4h			long	load, store, alux, alux
MOVSD mem16, mem16	A5h			long	load, store, alu, alu
MOVSW mem32, mem32	A5h			long	load, store, alu, alu
MOVSX reg16/32, mreg8	0Fh	BEh	11-xxx-xxx	short	alu
MOVSX reg16/32, mem8	0Fh	BEh	mm-xxx-xxx	short	load, alu
MOVSX reg32, mreg16	0Fh	BFh	11-xxx-xxx	short	alu
MOVSX reg32, mem16	0Fh	BFh	mm-xxx-xxx	short	load, alu
MOVZX reg16/32, mreg8	0Fh	B6h	11-xxx-xxx	short	alu
MOVZX reg16/32, mem8	0Fh	B6h	mm-xxx-xxx	short	load, alu
MOVZX reg32, mreg16	0Fh	B7h	11-xxx-xxx	short	alu
MOVZX reg32, mem16	0Fh	B7h	mm-xxx-xxx	short	load, alu
MUL AL, mreg8	F6h		11-100-xxx	vector	
MUL AL, mem8	F6h		mm-100-xxx	vector	
MUL EAX, mreg16/32	F7h		11-100-xxx	vector	
MUL EAX, mem16/32	F7h		mm-100-xxx	vector	
NEG mreg8	F6h		11-011-xxx	short	alux

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
NEG mem8	F6h		mm-011-xxx	vector	
NEG mreg16/32	F7h		11-011-xxx	short	alu
NEG mem16/32	F7h		mm-011-xxx	vector	
NOP (XCHG AX, AX)	90h			short	limm
NOT mreg8	F6h		11-010-xxx	short	alux
NOT mem8	F6h		mm-010-xxx	vector	
NOT mreg16/32	F7h		11-010-xxx	short	alu
NOT mem16/32	F7h		mm-010-xxx	vector	
OR mreg8, reg8	08h		11-xxx-xxx	short	alux
OR mem8, reg8	08h		mm-xxx-xxx	long	load, alux, store
OR mreg16/32, reg16/32	09h		11-xxx-xxx	short	alu
OR mem16/32, reg16/32	09h		mm-xxx-xxx	long	load, alu, store
OR reg8, mreg8	0Ah		11-xxx-xxx	short	alux
OR reg8, mem8	0Ah		mm-xxx-xxx	short	load, alux
OR reg16/32, mreg16/32	0Bh		11-xxx-xxx	short	alu
OR reg16/32, mem16/32	0Bh		mm-xxx-xxx	short	load, alu
OR AL, imm8	0Ch			short	alux
OR EAX, imm16/32	0Dh			short	alu
OR mreg8, imm8	80h		11-001-xxx	short	alux
OR mem8, imm8	80h		mm-001-xxx	long	load, alux, store
OR mreg16/32, imm16/32	81h		11-001-xxx	short	alu
OR mem16/32, imm16/32	81h		mm-001-xxx	long	load, alu, store
OR mreg16/32, imm8 (signed ext.)	83h		11-001-xxx	short	alux
OR mem16/32, imm8 (signed ext.)	83h		mm-001-xxx	long	load, alux, store
OUT imm8, AL	E6h			vector	
OUT imm8, AX	E7h			vector	
OUT imm8, EAX	E7h			vector	
OUT DX, AL	EEh			vector	
OUT DX, AX	EFh			vector	
OUT DX, EAX	EFh			vector	
POP ES	07h			vector	
POP SS	17h			vector	
POP DS	1Fh			vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
POP FS	0Fh	A1h		vector	
POP GS	0Fh	A9h		vector	
POP EAX	58h			short	load, alu
POP ECX	59h			short	load, alu
POP EDX	5Ah			short	load, alu
POP EBX	5Bh			short	load, alu
POP ESP	5Ch			short	load, alu
POP EBP	5Dh			short	load, alu
POP ESI	5Eh			short	load, alu
POP EDI	5Fh			short	load, alu
POP mreg 16/32	8Fh		11-000-xxx	short	load, alu
POP mem 16/32	8Fh		mm-000-xxx	long	load, store, alu
POPA/POPAD	61h			vector	
POPF/POPFD	9Dh			vector	
PUSH ES	06h			long	load, store
PUSH CS	0Eh			vector	
PUSH FS	0Fh	A0h		vector	
PUSH GS	0Fh	A8h		vector	
PUSH SS	16h			vector	
PUSH DS	1Eh			long	load, store
PUSH EAX	50h			short	store
PUSH ECX	51h			short	store
PUSH EDX	52h			short	store
PUSH EBX	53h			short	store
PUSH ESP	54h			short	store
PUSH EBP	55h			short	store
PUSH ESI	56h			short	store
PUSH EDI	57h			short	store
PUSH imm8	6Ah			long	store
PUSH imm16/32	68h			long	store
PUSH mreg16/32	FFh		11-110-xxx	vector	
PUSH mem16/32	FFh		mm-110-xxx	long	load, store
PUSHA/PUSHAD	60h			vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
PUSHF/PUSHFD	9Ch			vector	
RCL mreg8, imm8	C0h		11-010-xxx	vector	
RCL mem8, imm8	C0h		mm-010-xxx	vector	
RCL mreg16/32, imm8	C1h		11-010-xxx	vector	
RCL mem16/32, imm8	C1h		mm-010-xxx	vector	
RCL mreg8, 1	D0h		11-010-xxx	vector	
RCL mem8, 1	D0h		mm-010-xxx	vector	
RCL mreg16/32, 1	D1h		11-010-xxx	vector	
RCL mem16/32, 1	D1h		mm-010-xxx	vector	
RCL mreg8, CL	D2h		11-010-xxx	vector	
RCL mem8, CL	D2h		mm-010-xxx	vector	
RCL mreg16/32, CL	D3h		11-010-xxx	vector	
RCL mem16/32, CL	D3h		mm-010-xxx	vector	
RCR mreg8, imm8	C0h		11-011-xxx	vector	
RCR mem8, imm8	C0h		mm-011-xxx	vector	
RCR mreg16/32, imm8	C1h		11-011-xxx	vector	
RCR mem16/32, imm8	C1h		mm-011-xxx	vector	
RCR mreg8, 1	D0h		11-011-xxx	vector	
RCR mem8, 1	D0h		mm-011-xxx	vector	
RCR mreg16/32, 1	D1h		11-011-xxx	vector	
RCR mem16/32, 1	D1h		mm-011-xxx	vector	
RCR mreg8, CL	D2h		11-011-xxx	vector	
RCR mem8, CL	D2h		mm-011-xxx	vector	
RCR mreg16/32, CL	D3h		11-011-xxx	vector	
RCR mem16/32, CL	D3h		mm-011-xxx	vector	
RET near imm16	C2h			vector	
RET near	C3h			vector	
RET far imm16	CAh			vector	
RET far	CBh			vector	
ROL mreg8, imm8	C0h		11-000-xxx	vector	
ROL mem8, imm8	C0h		mm-000-xxx	vector	
ROL mreg16/32, imm8	C1h		11-000-xxx	vector	
ROL mem16/32, imm8	C1h		mm-000-xxx	vector	

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
ROL mreg8, 1	D0h		11-000-xxx	vector	
ROL mem8, 1	D0h		mm-000-xxx	vector	
ROL mreg16/32, 1	D1h		11-000-xxx	vector	
ROL mem16/32, 1	D1h		mm-000-xxx	vector	
ROL mreg8, CL	D2h		11-000-xxx	vector	
ROL mem8, CL	D2h		mm-000-xxx	vector	
ROL mreg16/32, CL	D3h		11-000-xxx	vector	
ROL mem16/32, CL	D3h		mm-000-xxx	vector	
ROR mreg8, imm8	C0h		11-001-xxx	vector	
ROR mem8, imm8	C0h		mm-001-xxx	vector	
ROR mreg16/32, imm8	C1h		11-001-xxx	vector	
ROR mem16/32, imm8	C1h		mm-001-xxx	vector	
ROR mreg8, 1	D0h		11-001-xxx	vector	
ROR mem8, 1	D0h		mm-001-xxx	vector	
ROR mreg16/32, 1	D1h		11-001-xxx	vector	
ROR mem16/32, 1	D1h		mm-001-xxx	vector	
ROR mreg8, CL	D2h		11-001-xxx	vector	
ROR mem8, CL	D2h		mm-001-xxx	vector	
ROR mreg16/32, CL	D3h		11-001-xxx	vector	
ROR mem16/32, CL	D3h		mm-001-xxx	vector	
SAHF	9Eh			vector	
SAR mreg8, imm8	C0h		11-111-xxx	short	alux
SAR mem8, imm8	C0h		mm-111-xxx	vector	
SAR mreg16/32, imm8	C1h		11-111-xxx	short	alu
SAR mem16/32, imm8	C1h		mm-111-xxx	vector	
SAR mreg8, 1	D0h		11-111-xxx	short	alux
SAR mem8, 1	D0h		mm-111-xxx	vector	
SAR mreg16/32, 1	D1h		11-111-xxx	short	alu
SAR mem16/32, 1	D1h		mm-111-xxx	vector	
SAR mreg8, CL	D2h		11-111-xxx	short	alux
SAR mem8, CL	D2h		mm-111-xxx	vector	
SAR mreg16/32, CL	D3h		11-111-xxx	short	alu
SAR mem16/32, CL	D3h		mm-111-xxx	vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
SBB mreg8, reg8	18h		11-xxx-xxx	vector	
SBB mem8, reg8	18h		mm-xxx-xxx	vector	
SBB mreg16/32, reg16/32	19h		11-xxx-xxx	vector	
SBB mem16/32, reg16/32	19h		mm-xxx-xxx	vector	
SBB reg8, mreg8	1Ah		11-xxx-xxx	vector	
SBB reg8, mem8	1Ah		mm-xxx-xxx	vector	
SBB reg16/32, mreg16/32	1Bh		11-xxx-xxx	vector	
SBB reg16/32, mem16/32	1Bh		mm-xxx-xxx	vector	
SBB AL, imm8	1Ch			vector	
SBB EAX, imm16/32	1Dh			vector	
SBB mreg8, imm8	80h		11-011-xxx	vector	
SBB mem8, imm8	80h		mm-011-xxx	vector	
SBB mreg16/32, imm16/32	81h		11-011-xxx	vector	
SBB mem16/32, imm16/32	81h		mm-011-xxx	vector	
SBB mreg8, imm8 (signed ext.)	83h		11-011-xxx	vector	
SBB mem8, imm8 (signed ext.)	83h		mm-011-xxx	vector	
SCASB AL, mem8	AEh			vector	
SCASW AX, mem16	AFh			vector	
SCASD EAX, mem32	AFh			vector	
SETO mreg8	0Fh	90h	11-xxx-xxx	vector	
SETO mem8	0Fh	90h	mm-xxx-xxx	vector	
SETNO mreg8	0Fh	91h	11-xxx-xxx	vector	
SETNO mem8	0Fh	91h	mm-xxx-xxx	vector	
SETB/SETNAE mreg8	0Fh	92h	11-xxx-xxx	vector	
SETB/SETNAE mem8	0Fh	92h	mm-xxx-xxx	vector	
SETNB/SETAE mreg8	0Fh	93h	11-xxx-xxx	vector	
SETNB/SETAE mem8	0Fh	93h	mm-xxx-xxx	vector	
SETZ/SETE mreg8	0Fh	94h	11-xxx-xxx	vector	
SETZ/SETE mem8	0Fh	94h	mm-xxx-xxx	vector	
SETNZ/SETNE mreg8	0Fh	95h	11-xxx-xxx	vector	
SETNZ/SETNE mem8	0Fh	95h	mm-xxx-xxx	vector	
SETBE/SETNA mreg8	0Fh	96h	11-xxx-xxx	vector	
SETBE/SETNA mem8	0Fh	96h	mm-xxx-xxx	vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
SETNBE/SETA mreg8	0Fh	97h	11-xxx-xxx	vector	
SETNBE/SETA mem8	0Fh	97h	mm-xxx-xxx	vector	
SETS mreg8	0Fh	98h	11-xxx-xxx	vector	
SETS mem8	0Fh	98h	mm-xxx-xxx	vector	
SETNS mreg8	0Fh	99h	11-xxx-xxx	vector	
SETNS mem8	0Fh	99h	mm-xxx-xxx	vector	
SETP/SETPE mreg8	0Fh	9Ah	11-xxx-xxx	vector	
SETP/SETPE mem8	0Fh	9Ah	mm-xxx-xxx	vector	
SETNP/SETPO mreg8	0Fh	9Bh	11-xxx-xxx	vector	
SETNP/SETPO mem8	0Fh	9Bh	mm-xxx-xxx	vector	
SETL/SETNGE mreg8	0Fh	9Ch	11-xxx-xxx	vector	
SETL/SETNGE mem8	0Fh	9Ch	mm-xxx-xxx	vector	
SETNL/SETGE mreg8	0Fh	9Dh	11-xxx-xxx	vector	
SETNL/SETGE mem8	0Fh	9Dh	mm-xxx-xxx	vector	
SETLE/SETNG mreg8	0Fh	9Eh	11-xxx-xxx	vector	
SETLE/SETNG mem8	0Fh	9Eh	mm-xxx-xxx	vector	
SETNLE/SETG mreg8	0Fh	9Fh	11-xxx-xxx	vector	
SETNLE/SETG mem8	0Fh	9Fh	mm-xxx-xxx	vector	
SGDT mem48	0Fh	01h	mm-000-xxx	vector	
SIDT mem48	0Fh	01h	mm-001-xxx	vector	
SHL/SAL mreg8, imm8	C0h		11-100-xxx	short	alux
SHL/SAL mem8, imm8	C0h		mm-100-xxx	vector	
SHL/SAL mreg16/32, imm8	C1h		11-100-xxx	short	alu
SHL/SAL mem16/32, imm8	C1h		mm-100-xxx	vector	
SHL/SAL mreg8, 1	D0h		11-100-xxx	short	alux
SHL/SAL mem8, 1	D0h		mm-100-xxx	vector	
SHL/SAL mreg16/32, 1	D1h		11-100-xxx	short	alu
SHL/SAL mem16/32, 1	D1h		mm-100-xxx	vector	
SHL/SAL mreg8, CL	D2h		11-100-xxx	short	alux
SHL/SAL mem8, CL	D2h		mm-100-xxx	vector	
SHL/SAL mreg16/32, CL	D3h		11-100-xxx	short	alu
SHL/SAL mem16/32, CL	D3h		mm-100-xxx	vector	
SHR mreg8, imm8	C0h		11-101-xxx	short	alux

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
SHR mem8, imm8	C0h		mm-101-xxx	vector	
SHR mreg16/32, imm8	C1h		11-101-xxx	short	alu
SHR mem16/32, imm8	C1h		mm-101-xxx	vector	
SHR mreg8, 1	D0h		11-101-xxx	short	alux
SHR mem8, 1	D0h		mm-101-xxx	vector	
SHR mreg16/32, 1	D1h		11-101-xxx	short	alu
SHR mem16/32, 1	D1h		mm-101-xxx	vector	
SHR mreg8, CL	D2h		11-101-xxx	short	alux
SHR mem8, CL	D2h		mm-101-xxx	vector	
SHR mreg16/32, CL	D3h		11-101-xxx	short	alu
SHR mem16/32, CL	D3h		mm-101-xxx	vector	
SHLD mreg16/32, reg16/32, imm8	0Fh	A4h	11-xxx-xxx	vector	
SHLD mem16/32, reg16/32, imm8	0Fh	A4h	mm-xxx-xxx	vector	
SHLD mreg16/32, reg16/32, CL	0Fh	A5h	11-xxx-xxx	vector	
SHLD mem16/32, reg16/32, CL	0Fh	A5h	mm-xxx-xxx	vector	
SHRD mreg16/32, reg16/32, imm8	0Fh	ACh	11-xxx-xxx	vector	
SHRD mem16/32, reg16/32, imm8	0Fh	ACh	mm-xxx-xxx	vector	
SHRD mreg16/32, reg16/32, CL	0Fh	ADh	11-xxx-xxx	vector	
SHRD mem16/32, reg16/32, CL	0Fh	ADh	mm-xxx-xxx	vector	
SLDT mreg16	0Fh	00h	11-000-xxx	vector	
SLDT mem16	0Fh	00h	mm-000-xxx	vector	
SMSW mreg16	0Fh	01h	11-100-xxx	vector	
SMSW mem16	0Fh	01h	mm-100-xxx	vector	
STC	F9h			vector	
STD	FDh			vector	
STI	FBh			vector	
STOSB mem8, AL	AAh			long	store, alux
STOSW mem16, AX	ABh			long	store, alu
STOSD mem32, EAX	ABh			long	store, alu
STR mreg16	0Fh	00h	11-001-xxx	vector	
STR mem16	0Fh	00h	mm-001-xxx	vector	
SUB mreg8, reg8	28h		11-xxx-xxx	short	alux
SUB mem8, reg8	28h		mm-xxx-xxx	long	load, alux, store

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
SUB mreg16/32, reg16/32	29h		11-xxx-xxx	short	alu
SUB mem16/32, reg16/32	29h		mm-xxx-xxx	long	load, alu, store
SUB reg8, mreg8	2Ah		11-xxx-xxx	short	alux
SUB reg8, mem8	2Ah		mm-xxx-xxx	short	load, alux
SUB reg16/32, mreg16/32	2Bh		11-xxx-xxx	short	alu
SUB reg16/32, mem16/32	2Bh		mm-xxx-xxx	short	load, alu
SUB AL, imm8	2Ch			short	alux
SUB EAX, imm16/32	2Dh			short	alu
SUB mreg8, imm8	80h		11-101-xxx	short	alux
SUB mem8, imm8	80h		mm-101-xxx	long	load, alux, store
SUB mreg16/32, imm16/32	81h		11-101-xxx	short	alu
SUB mem16/32, imm16/32	81h		mm-101-xxx	long	load, alu, store
SUB mreg16/32, imm8 (signed ext.)	83h		11-101-xxx	short	alux
SUB mem16/32, imm8 (signed ext.)	83h		mm-101-xxx	long	load, alux, store
SYSCALL (only supported on AMD-K6-2 and AMD-K6-III processors)	0Fh	05h		vector	
SYSRET (only supported on AMD-K6-2 and AMD-K6-III processors)	0Fh	07h		vector	
TEST mreg8, reg8	84h		11-xxx-xxx	short	alux
TEST mem8, reg8	84h		mm-xxx-xxx	vector	
TEST mreg16/32, reg16/32	85h		11-xxx-xxx	short	alu
TEST mem16/32, reg16/32	85h		mm-xxx-xxx	vector	
TEST AL, imm8	A8h			long	alux
TEST EAX, imm16/32	A9h			long	alu
TEST mreg8, imm8	F6h		11-000-xxx	long	alux
TEST mem8, imm8	F6h		mm-000-xxx	long	load, alux
TEST mreg16/32, imm16/32	F7h		11-000-xxx	long	alu
TEST mem16/32, imm16/32	F7h		mm-000-xxx	long	load, alu
VERR mreg16	0Fh	00h	11-100-xxx	vector	
VERR mem16	0Fh	00h	mm-100-xxx	vector	
VERW mreg16	0Fh	00h	11-101-xxx	vector	
VERW mem16	0Fh	00h	mm-101-xxx	vector	
WAIT	9Bh			vector	

Table 6.   Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
WBINVD	0Fh	09h		vector	
XADD mreg8, reg8	0Fh	C0h	11-100-xxx	vector	
XADD mem8, reg8	0Fh	C0h	mm-100-xxx	vector	
XADD mreg16/32, reg16/32	0Fh	C1h	11-101-xxx	vector	
XADD mem16/32, reg16/32	0Fh	C1h	mm-101-xxx	vector	
XCHG reg8, mreg8	86h		11-xxx-xxx	vector	
XCHG reg8, mem8	86h		mm-xxx-xxx	vector	
XCHG reg16/32, mreg16/32	87h		11-xxx-xxx	vector	
XCHG reg16/32, mem16/32	87h		mm-xxx-xxx	vector	
XCHG EAX, EAX	90h			short	limm
XCHG EAX, ECX	91h			long	alu, alu, alu
XCHG EAX, EDX	92h			long	alu, alu, alu
XCHG EAX, EBX	93h			long	alu, alu, alu
XCHG EAX, ESP	94h			long	alu, alu, alu
XCHG EAX, EBP	95h			long	alu, alu, alu
XCHG EAX, ESI	96h			long	alu, alu, alu
XCHG EAX, EDI	97h			long	alu, alu, alu
XLAT	D7h			vector	
XOR mreg8, reg8	30h		11-xxx-xxx	short	alux
XOR mem8, reg8	30h		mm-xxx-xxx	long	load, alux, store
XOR mreg16/32, reg16/32	31h		11-xxx-xxx	short	alu
XOR mem16/32, reg16/32	31h		mm-xxx-xxx	long	load, alu, store
XOR reg8, mreg8	32h		11-xxx-xxx	short	alux
XOR reg8, mem8	32h		mm-xxx-xxx	short	load, alux
XOR reg16/32, mreg16/32	33h		11-xxx-xxx	short	alu
XOR reg16/32, mem16/32	33h		mm-xxx-xxx	short	load, alu
XOR AL, imm8	34h			short	alux
XOR EAX, imm16/32	35h			short	alu
XOR mreg8, imm8	80h		11-110-xxx	short	alux
XOR mem8, imm8	80h		mm-110-xxx	long	load, alux, store
XOR mreg16/32, imm16/32	81h		11-110-xxx	short	alu
XOR mem16/32, imm16/32	81h		mm-110-xxx	long	load, alu, store

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations
XOR mreg16/32, imm8 (signed ext.)	83h		11-110-xxx	short	alux
XOR mem16/32, imm8 (signed ext.)	83h		mm-110-xxx	long	load, alux, store

Table 7. MMX™ Instructions

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
EMMS	0Fh	77h		vector		
MOVD mmreg, mreg32	0Fh	6Eh	11-xxx-xxx	short	meu	**
MOVD mmreg, mem32	0Fh	6Eh	mm-xxx-xxx	short	mload	
MOVD mreg32, mmreg	0Fh	7Eh	11-xxx-xxx	short	mstore, load	**
MOVD mem32, mmreg	0Fh	7Eh	mm-xxx-xxx	short	mstore	
MOVQ mmreg1, mmreg2	0Fh	6Fh	11-xxx-xxx	short	meu	
MOVQ mmreg, mem64	0Fh	6Fh	mm-xxx-xxx	short	mload	
MOVQ mmreg2, mmreg1	0Fh	7Fh	11-xxx-xxx	short	meu	
MOVQ mem64, mmreg	0Fh	7Fh	mm-xxx-xxx	short	mstore	
PACKSSDW mmreg1, mmreg2	0Fh	6Bh	11-xxx-xxx	short	meu	
PACKSSDW mmreg, mem64	0Fh	6Bh	mm-xxx-xxx	short	mload, meu	
PACKSSWB mmreg1, mmreg2	0Fh	63h	11-xxx-xxx	short	meu	
PACKSSWB mmreg, mem64	0Fh	64h	mm-xxx-xxx	short	mload, meu	
PACKUSWB mmreg1, mmreg2	0Fh	67h	11-xxx-xxx	short	meu	
PACKUSWB mmreg, mem64	0Fh	67h	mm-xxx-xxx	short	mload, meu	
PADDB mmreg1, mmreg2	0Fh	FCh	11-xxx-xxx	short	meu	
PADDB mmreg, mem64	0Fh	FCh	mm-xxx-xxx	short	mload, meu	
PADDD mmreg1, mmreg2	0Fh	FEh	11-xxx-xxx	short	meu	
PADDD mmreg, mem64	0Fh	FEh	mm-xxx-xxx	short	mload, meu	
PADDSB mmreg1, mmreg2	0Fh	ECh	11-xxx-xxx	short	meu	
PADDSB mmreg, mem64	0Fh	ECh	mm-xxx-xxx	short	mload, meu	
PADDSW mmreg1, mmreg2	0Fh	EDh	11-xxx-xxx	short	meu	
PADDSW mmreg, mem64	0Fh	EDh	mm-xxx-xxx	short	mload, meu	
PADDUSB mmreg1, mmreg2	0Fh	DCh	11-xxx-xxx	short	meu	
PADDUSB mmreg, mem64	0Fh	DCh	mm-xxx-xxx	short	mload, meu	

Notes:
    ** *Bits 2, 1, and 0 of the modR/M byte select the integer register.*

Table 7.   MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
PADDUSW mmreg1, mmreg2	0Fh	DDh	11-xxx-xxx	short	meu	
PADDUSW mmreg, mem64	0Fh	DDh	mm-xxx-xxx	short	mload, meu	
PADDW mmreg1, mmreg2	0Fh	FDh	11-xxx-xxx	short	meu	
PADDW mmreg, mem64	0Fh	FDh	mm-xxx-xxx	short	mload, meu	
PAND mmreg1, mmreg2	0Fh	DBh	11-xxx-xxx	short	meu	
PAND mmreg, mem64	0Fh	DBh	mm-xxx-xxx	short	mload, meu	
PANDN mmreg1, mmreg2	0Fh	DFh	11-xxx-xxx	short	meu	
PANDN mmreg, mem64	0Fh	DFh	mm-xxx-xxx	short	mload, meu	
PCMPEQB mmreg1, mmreg2	0Fh	74h	11-xxx-xxx	short	meu	
PCMPEQB mmreg, mem64	0Fh	74h	mm-xxx-xxx	short	mload, meu	
PCMPEQD mmreg1, mmreg2	0Fh	76h	11-xxx-xxx	short	meu	
PCMPEQD mmreg, mem64	0Fh	76h	mm-xxx-xxx	short	mload, meu	
PCMPEQW mmreg1, mmreg2	0Fh	75h	11-xxx-xxx	short	meu	
PCMPEQW mmreg, mem64	0Fh	75h	mm-xxx-xxx	short	mload, meu	
PCMPGTB mmreg1, mmreg2	0Fh	64h	11-xxx-xxx	short	meu	
PCMPGTB mmreg, mem64	0Fh	64h	mm-xxx-xxx	short	mload, meu	
PCMPGTD mmreg1, mmreg2	0Fh	66h	11-xxx-xxx	short	meu	
PCMPGTD mmreg, mem64	0Fh	66h	mm-xxx-xxx	short	mload, meu	
PCMPGTW mmreg1, mmreg2	0Fh	65h	11-xxx-xxx	short	meu	
PCMPGTW mmreg, mem64	0Fh	65h	mm-xxx-xxx	short	mload, meu	
PMADDWD mmreg1, mmreg2	0Fh	F5h	11-xxx-xxx	short	meu	
PMADDWD mmreg, mem64	0Fh	F5h	mm-xxx-xxx	short	mload, meu	
PMULHW mmreg1, mmreg2	0Fh	E5h	11-xxx-xxx	short	meu	
PMULHW mmreg, mem64	0Fh	E5h	mm-xxx-xxx	short	mload, meu	
PMULLW mmreg1, mmreg2	0Fh	D5h	11-xxx-xxx	short	meu	
PMULLW mmreg, mem64	0Fh	D5h	mm-xxx-xxx	short	mload, meu	
POR mmreg1, mmreg2	0Fh	EBh	11-xxx-xxx	short	meu	
POR mmreg, mem64	0Fh	EBh	mm-xxx-xxx	short	mload, meu	
PSLLW mmreg1, mmreg2	0Fh	F1h	11-xxx-xxx	short	meu	
PSLLW mmreg, mem64	0Fh	F1h	mm-xxx-xxx	short	mload, meu	
PSLLW mmreg, imm8	0Fh	71h	11-110-xxx	short	meu	

Notes:

   ** Bits 2, 1, and 0 of the modR/M byte select the integer register.

Table 7.   MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
PSLLD mmreg1, mmreg2	0Fh	F2h	11-xxx-xxx	short	meu	
PSLLD mmreg, mem64	0Fh	F2h	mm-xxx-xxx	short	mload, meu	
PSLLD mmreg, imm8	0Fh	72h	11-110-xxx	short	meu	
PSLLQ mmreg1, mmreg2	0Fh	F3h	11-xxx-xxx	short	meu	
PSLLQ mmreg, mem64	0Fh	F3h	mm-xxx-xxx	short	mload, meu	
PSLLQ mmreg, imm8	0Fh	73h	11-110-xxx	short	meu	
PSRAW mmreg1, mmreg2	0Fh	E1h	11-xxx-xxx	short	meu	
PSRAW mmreg, mem64	0Fh	E1h	mm-xxx-xxx	short	mload, meu	
PSRAW mmreg, imm8	0Fh	71h	11-100-xxx	short	meu	
PSRAD mmreg1, mmreg2	0Fh	E2h	11-xxx-xxx	short	meu	
PSRAD mmreg, mem64	0Fh	E2h	mm-xxx-xxx	short	mload, meu	
PSRAD mmreg, imm8	0Fh	72h	11-100-xxx	short	meu	
PSRLW mmreg1, mmreg2	0Fh	D1h	11-xxx-xxx	short	meu	
PSRLW mmreg, mem64	0Fh	D1h	mm-xxx-xxx	short	mload, meu	
PSRLW mmreg, imm8	0Fh	71h	11-010-xxx	short	meu	
PSRLD mmreg1, mmreg2	0Fh	D2h	11-xxx-xxx	short	meu	
PSRLD mmreg, mem64	0Fh	D2h	mm-xxx-xxx	short	mload, meu	
PSRLD mmreg, imm8	0Fh	72h	11-010-xxx	short	meu	
PSRLQ mmreg1, mmreg2	0Fh	D3h	11-xxx-xxx	short	meu	
PSRLQ mmreg, mem64	0Fh	D3h	mm-xxx-xxx	short	mload, meu	
PSRLQ mmreg, imm8	0Fh	73h	11-010-xxx	short	meu	
PSUBB mmreg1, mmreg2	0Fh	F8h	11-xxx-xxx	short	meu	
PSUBB mmreg, mem64	0Fh	F8h	mm-xxx-xxx	short	mload, meu	
PSUBD mmreg1, mmreg2	0Fh	FAh	11-xxx-xxx	short	meu	
PSUBD mmreg, mem64	0Fh	FAh	mm-xxx-xxx	short	mload, meu	
PSUBSB mmreg1, mmreg2	0Fh	E8h	11-xxx-xxx	short	meu	
PSUBSB mmreg, mem64	0Fh	E8h	mm-xxx-xxx	short	mload, meu	
PSUBSW mmreg1, mmreg2	0Fh	E9h	11-xxx-xxx	short	meu	
PSUBSW mmreg, mem64	0Fh	E9h	mm-xxx-xxx	short	mload, meu	
PSUBUSB mmreg1, mmreg2	0Fh	D8h	11-xxx-xxx	short	meu	
PSUBUSB mmreg, mem64	0Fh	D8h	mm-xxx-xxx	short	mload, meu	

Notes:
  ** *Bits 2, 1, and 0 of the modR/M byte select the integer register.*

Table 7.   MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
PSUBUSW mmreg1, mmreg2	0Fh	D9h	11-xxx-xxx	short	meu	
PSUBUSW mmreg, mem64	0Fh	D9h	mm-xxx-xxx	short	mload, meu	
PSUBW mmreg1, mmreg2	0Fh	F9h	11-xxx-xxx	short	meu	
PSUBW mmreg, mem64	0Fh	F9h	mm-xxx-xxx	short	mload, meu	
PUNPCKHBW mmreg1, mmreg2	0Fh	68h	11-xxx-xxx	short	meu	
PUNPCKHBW mmreg, mem64	0Fh	68h	mm-xxx-xxx	short	mload, meu	
PUNPCKHWD mmreg1, mmreg2	0Fh	69h	11-xxx-xxx	short	meu	
PUNPCKHWD mmreg, mem64	0Fh	69h	mm-xxx-xxx	short	mload, meu	
PUNPCKHDQ mmreg1, mmreg2	0Fh	6Ah	11-xxx-xxx	short	meu	
PUNPCKHDQ mmreg, mem64	0Fh	6Ah	mm-xxx-xxx	short	mload, meu	
PUNPCKLBW mmreg1, mmreg2	0Fh	60h	11-xxx-xxx	short	meu	
PUNPCKLBW mmreg, mem64	0Fh	60h	mm-xxx-xxx	short	mload, meu	
PUNPCKLWD mmreg1, mmreg2	0Fh	61h	11-xxx-xxx	short	meu	
PUNPCKLWD mmreg, mem64	0Fh	61h	mm-xxx-xxx	short	mload, meu	
PUNPCKLDQ mmreg1, mmreg2	0Fh	62h	11-xxx-xxx	short	meu	
PUNPCKLDQ mmreg, mem64	0Fh	62h	mm-xxx-xxx	short	mload, meu	
PXOR mmreg1, mmreg2	0Fh	EFh	11-xxx-xxx	short	meu	
PXOR mmreg, mem64	0Fh	EFh	mm-xxx-xxx	short	mload, meu	

Notes:
***   **   Bits 2, 1, and 0 of the modR/M byte select the integer register.***

Table 8.   Floating-Point Instructions

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
F2XM1	D9h	F0h		short	float	
FABS	D9h	F1h		short	float	
FADD ST(0), ST(i)	D8h		11-000-xxx	short	float	*
FADD ST(0), mem32real	D8h		mm-000-xxx	short	fload, float	
FADD ST(i), ST(0)	DCh		11-000-xxx	short	float	*
FADD ST(0), mem64real	DCh		mm-000-xxx	short	fload, float	
FADDP ST(i), ST(0)	DEh		11-000-xxx	short	float	*
FBLD	DFh		mm-100-xxx	vector		
FBSTP	DFh		mm-110-xxx	vector		
FCHS	D9h	E0h		short	float	
FCLEX	DBh	E2h		vector		
FCOM ST(0), ST(i)	D8h		11-010-xxx	short	float	*
FCOM ST(0), mem32real	D8h		mm-010-xxx	short	fload, float	
FCOM ST(0), mem64real	DCh		mm-010-xxx	short	fload, float	
FCOMP ST(0), ST(i)	D8h		11-011-xxx	short	float	*
FCOMP ST(0), mem32real	D8h		mm-011-xxx	short	fload, float	
FCOMP ST(0), mem64real	DCh		mm-011-xxx	short	fload, float	
FCOMPP	DEh	D9h	11-011-001	short	float	
FCOS	D9h	FFh		short	float	
FDECSTP	D9h	F6h		short	float	
FDIV ST(0), ST(i) (single precision)	D8h		11-110-xxx	short	float	*
FDIV ST(0), ST(i) (double precision)	D8h		11-110-xxx	short	float	*
FDIV ST(0), ST(i) (extended precision)	D8h		11-110-xxx	short	float	*
FDIV ST(i), ST(0) (single precision)	DCh		11-111-xxx	short	float	*
FDIV ST(i), ST(0) (double precision)	DCh		11-111-xxx	short	float	*
FDIV ST(i), ST(0) (extended precision)	DCh		11-111-xxx	short	float	*
FDIV ST(0), mem32real	D8h		mm-110-xxx	short	fload, float	
FDIV ST(0), mem64real	DCh		mm-110-xxx	short	fload, float	
FDIVP ST(0), ST(i)	DEh		11-111-xxx	short	float	*
FDIVR ST(0), ST(i)	D8h		11-110-xxx	short	float	*
FDIVR ST(i), ST(0)	DCh		11-111-xxx	short	float	*

Notes:
  *   *The last three bits of the modR/M byte select the stack entry ST(i).*

Table 8.   Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
FDIVR ST(0), mem32real	D8h		mm-111-xxx	short	fload, float	
FDIVR ST(0), mem64real	DCh		mm-111-xxx	short	fload, float	
FDIVRP ST(i), ST(0)	DEh		11-110-xxx	short	float	*
FFREE ST(i)	DDh		11-000-xxx	short	float	*
FIADD ST(0), mem32int	DAh		mm-000-xxx	short	fload, float	
FIADD ST(0), mem16int	DEh		mm-000-xxx	short	fload, float	
FICOM ST(0), mem32int	DAh		mm-010-xxx	short	fload, float	
FICOM ST(0), mem16int	DEh		mm-010-xxx	short	fload, float	
FICOMP ST(0), mem32int	DAh		mm-011-xxx	short	fload, float	
FICOMP ST(0), mem16int	DEh		mm-011-xxx	short	fload, float	
FIDIV ST(0), mem32int	DAh		mm-110-xxx	short	fload, float	
FIDIV ST(0), mem16int	DEh		mm-110-xxx	short	fload, float	
FIDIVR ST(0), mem32int	DAh		mm-111-xxx	short	fload, float	
FIDIVR ST(0), mem16int	DEh		mm-111-xxx	short	fload, float	
FILD mem16int	DFh		mm-000-xxx	short	fload, float	
FILD mem32int	DBh		mm-000-xxx	short	fload, float	
FILD mem64int	DFh		mm-101-xxx	short	fload, float	
FIMUL ST(0), mem32int	DAh		mm-001-xxx	short	fload, float	
FIMUL ST(0), mem16int	DEh		mm-001-xxx	short	fload, float	
FINCSTP	D9h	F7h		short	float	
FINIT	DBh	E3h		vector		
FIST mem16int	DFh		mm-010-xxx	short	fload, float	
FIST mem32int	DBh		mm-010-xxx	short	fload, float	
FISTP mem16int	DFh		mm-011-xxx	short	fload, float	
FISTP mem32int	DBh		mm-011-xxx	short	fload, float	
FISTP mem64int	DFh		mm-111-xxx	short	fload, float	
FISUB ST(0), mem32int	DAh		mm-100-xxx	short	fload, float	
FISUB ST(0), mem16int	DEh		mm-100-xxx	short	fload, float	
FISUBR ST(0), mem32int	DAh		mm-101-xxx	short	fload, float	
FISUBR ST(0), mem16int	DEh		mm-101-xxx	short	fload, float	
FLD ST(i)	D9h		11-000-xxx	short	fload, float	*

Notes:
  *   The last three bits of the modR/M byte select the stack entry ST(i).

Table 8. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
FLD mem32real	D9h		mm-000-xxx	short	fload, float	
FLD mem64real	DDh		mm-000-xxx	short	fload, float	
FLD mem80real	DBh		mm-101-xxx	vector		
FLD1	D9h	E8h		short	fload, float	
FLDCW	D9h		mm-101-xxx	vector		
FLDENV	D9h		mm-100-xxx	short	fload, float	
FLDL2E	D9h	EAh		short	float	
FLDL2T	D9h	E9h		short	float	
FLDLG2	D9h	ECh		short	float	
FLDLN2	D9h	EDh		short	float	
FLDPI	D9h	EBh		short	float	
FLDZ	D9h	EEh		short	float	
FMUL ST(0), ST(i)	D8h		11-001-xxx	short	float	*
FMUL ST(i), ST(0)	DCh		11-001-xxx	short	float	*
FMUL ST(0), mem32real	D8h		mm-001-xxx	short	fload, float	
FMUL ST(0), mem64real	DCh		mm-001-xxx	short	fload, float	
FMULP ST(0), ST(i)	DEh		11-001-xxx	short	float	*
FNOP	D9h	D0h		short	float	
FPATAN	D9h	F3h		short	float	
FPREM	D9h	F8h		short	float	
FPREM1	D9h	F5h		short	float	
FPTAN	D9h	F2h		vector		
FRNDINT	D9h	FCh		short	float	
FRSTOR	DDh		mm-100-xxx	vector		
FSAVE	DDh		mm-110-xxx	vector		
FSCALE	D9h	FDh		short	float	
FSIN	D9h	FEh		short	float	
FSINCOS	D9h	FBh		vector		
FSQRT (single precision)	D9h	FAh		short	float	
FSQRT (double precision)	D9h	FAh		short	float	
FSQRT (extended precision)	D9h	FAh		short	float	

Notes:
   * The last three bits of the modR/M byte select the stack entry ST(i).

Table 8.   Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
FST mem32real	D9h		mm-010-xxx	short	fstore	
FST mem64real	DDh		mm-010-xxx	short	fstore	
FST ST(i)	DDh		11-010-xxx	short	fstore	*
FSTCW	D9h		mm-111-xxx	vector		
FSTENV	D9h		mm-110-xxx	vector		
FSTP mem32real	D9h		mm-011-xxx	short	fstore	
FSTP mem64real	DDh		mm-011-xxx	short	fstore	
FSTP mem80real	D9h		mm-111-xxx	vector		
FSTP ST(i)	DDh		11-011-xxx	short	float	*
FSTSW AX	DFh	E0h		vector		
FSTSW mem16	DDh		mm-111-xxx	vector		
FSUB ST(0), mem32real	D8h		mm-100-xxx	short	fload, float	
FSUB ST(0), mem64real	DCh		mm-100-xxx	short	fload, float	
FSUB ST(0), ST(i)	D8h		11-100-xxx	short	float	*
FSUB ST(i), ST(0)	DCh		11-101-xxx	short	float	*
FSUBP ST(0), ST(i)	DEh		11-101-xxx	short	float	*
FSUBR ST(0), mem32real	D8h		mm-101-xxx	short	fload, float	
FSUBR ST(0), mem64real	DCh		mm-101-xxx	short	fload, float	
FSUBR ST(0), ST(i)	D8h		11-100-xxx	short	float	*
FSUBR ST(i), ST(0)	DCh		11-101-xxx	short	float	*
FSUBRP ST(i), ST(0)	DEh		11-100-xxx	short	float	*
FTST	D9h	E4h		short	float	
FUCOM	DDh		11-100-xxx	short	float	
FUCOMP	DDh		11-101-xxx	short	float	
FUCOMPP	DAh	E9h		short	float	
FXAM	D9h	E5h		short	float	
FXCH	D9h		11-001-xxx	short	float	
FXTRACT	D9h	F4h		vector		
FYL2X	D9h	F1h		short	float	
FYL2XP1	D9h	F9h		short	float	
FWAIT	9Bh			vector		

Notes:

  *   *The last three bits of the modR/M byte select the stack entry ST(i).*

Table 9.   3DNow!™ Instructions

Instruction Mnemonic	Prefix Byte(s)	Opcode Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
FEMMS	0Fh	0Eh		vector		1
PAVGUSB mmreg1, mmreg2	0Fh, 0Fh	BFh	11-xxx-xxx	short	meu	
PAVGUSB mmreg, mem64	0Fh, 0Fh	BFh	mm-xxx-xxx	short	mload, meu	
PFADD mmreg1, mmreg2	0Fh, 0Fh	9Eh	11-xxx-xxx	short	meu	
PFADD mmreg, mem64	0Fh, 0Fh	9Eh	mm-xxx-xxx	short	mload, meu	
PFSUB mmreg1, mmreg2	0Fh, 0Fh	9Ah	11-xxx-xxx	short	meu	
PFSUB mmreg, mem64	0Fh, 0Fh	9Ah	mm-xxx-xxx	short	mload, meu	
PFSUBR mmreg1, mmreg2	0Fh, 0Fh	AAh	11-xxx-xxx	short	meu	
PFSUBR mmreg, mem64	0Fh, 0Fh	AAh	mm-xxx-xxx	short	mload, meu	
PFACC mmreg1, mmreg2	0Fh, 0Fh	AEh	11-xxx-xxx	short	meu	
PFACC mmreg, mem64	0Fh, 0Fh	AEh	mm-xxx-xxx	short	mload, meu	
PFMUL mmreg1, mmreg2	0Fh, 0Fh	B4h	11-xxx-xxx	short	meu	
PFMUL mmreg, mem64	0Fh, 0Fh	B4h	mm-xxx-xxx	short	mload, meu	
PFCMPGE mmreg1, mmreg2	0Fh, 0Fh	90h	11-xxx-xxx	short	meu	
PFCMPGE mmreg, mem64	0Fh, 0Fh	90h	mm-xxx-xxx	short	mload, meu	
PFCMPGT mmreg1, mmreg2	0Fh, 0Fh	A0h	11-xxx-xxx	short	meu	
PFCMPGT mmreg, mem64	0Fh, 0Fh	A0h	mm-xxx-xxx	short	mload, meu	
PFCMPEQ mmreg1, mmreg2	0Fh, 0Fh	B0h	11-xxx-xxx	short	meu	
PFCMPEQ mmreg, mem64	0Fh, 0Fh	B0h	mm-xxx-xxx	short	mload, meu	
PFMIN mmreg1, mmreg2	0Fh, 0Fh	94h	11-xxx-xxx	short	meu	
PFMIN mmreg, mem64	0Fh, 0Fh	94H	mm-xxx-xxx	short	mload, meu	
PFMAX mmreg1, mmreg2	0Fh, 0Fh	A4h	11-xxx-xxx	short	meu	
PFMAX mmreg, mem64	0Fh, 0Fh	A4h	mm-xxx-xxx	short	mload, meu	
PI2FD mmreg1, mmreg2	0Fh, 0Fh	0Dh	11-xxx-xxx	short	meu	
PI2FD mmreg, mem64	0Fh, 0Fh	0Dh	mm-xxx-xxx	short	mload, meu	
PF2ID mmreg1, mmreg2	0Fh, 0Fh	1Dh	11-xxx-xxx	short	meu	
PF2ID mmreg, mem64	0Fh, 0Fh	1Dh	mm-xxx-xxx	short	mload, meu	
PFRCP mmreg1, mmreg2	0Fh, 0Fh	96h	11-xxx-xxx	short	meu	
PFRCP mmreg, mem64	0Fh, 0Fh	96h	mm-xxx-xxx	short	mload, meu	

Notes:

1. *For more information about the FEMMS instruction, see "AMD-K6®-2 and AMD-K6®-III Processors Multimedia Coding Optimizations" on page 67.*

2. *For PREFETCH and PREFETCHW, the mem8 value refers to an address in the 32-byte line that will be prefetched.*

3. *PREFETCHW will be implemented in a future K86™ processor. On the AMD-K6-2 processor, this instruction performs in the same manner as the PREFETCH instruction.*

Table 9.   3DNow!™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	Opcode Byte	ModR/M Byte	Decode Type	RISC86® Operations	Note
PFRSQRT mmreg1, mmreg2	0Fh, 0Fh	97h	11-xxx-xxx	short	meu	
PFRSQRT mmreg, mem64	0Fh, 0Fh	97h	mm-xxx-xxx	short	mload, meu	
PFRCPIT1 mmreg1, mmreg2	0Fh, 0Fh	A6h	11-xxx-xxx	short	meu	
PFRCPIT1 mmreg, mem64	0Fh, 0Fh	A6h	mm-xxx-xxx	short	mload, meu	
PFRSQIT1 mmreg1, mmreg2	0Fh, 0Fh	A7h	11-xxx-xxx	short	meu	
PFRSQIT1 mmreg, mem64	0Fh, 0Fh	A7h	mm-xxx-xxx	short	mload, meu	
PFRCPIT2 mmreg1, mmreg2	0Fh, 0Fh	B6h	11-xxx-xxx	short	meu	
PFRCPIT2 mmreg, mem64	0Fh, 0Fh	B6h	mm-xxx-xxx	short	mload, meu	
PMULHRW mmreg1, mmreg2	0Fh, 0Fh	B7h	11-xxx-xxx	short	meu	
PMULHRW mmreg1, mem64	0Fh, 0Fh	B7h	mm-xxx-xxx	short	mload, meu	
PREFETCH mem8	0Fh	0Dh	mm-000-xxx	vector	load	2
PREFETCHW mem8	0Fh	0Dh	mm-001-xxx	vector	load	2, 3

Notes:

1. *For more information about the FEMMS instruction, see "AMD-K6®-2 and AMD-K6®-III Processors Multimedia Coding Optimizations" on page 67.*

2. *For PREFETCH and PREFETCHW, the mem8 value refers to an address in the 32-byte line that will be prefetched.*

3. *PREFETCHW will be implemented in a future K86™ processor. On the AMD-K6-2 processor, this instruction performs in the same manner as the PREFETCH instruction.*

5

Optimization Coding Guidelines

General x86 Optimization Techniques

This section describes general code optimization techniques specific to superscalar processors (that is, techniques common to the AMD-K6 family of processors, AMD-K5™ processor, and Pentium family processors). In general, all optimization techniques used for the AMD-K5 processor, Pentium, and Pentium Pro processors either improve the performance of the AMD-K6 family or are not required and have neutral effect (usually due to fewer coding restrictions with the AMD-K6 family).

Short Forms—Use shorter forms of instructions to increase the effective number of instructions that can be examined for decoding at any one time. Use 8-bit displacements and jump offsets where possible.

Simple Instructions—Use simple instructions with hardwired decode (pairable, short, or fast) because they perform more efficiently. This includes "register←register op memory" as well as "register←register op register" forms of instructions.

Dependencies—Spread out true dependencies to increase the opportunities for parallel execution. Anti-dependencies and output dependencies do not impact performance.

Memory Operands—Instructions that operate on data in memory (load/operation/store) can inhibit parallelism. The use of separate move and ALU instructions allows better code scheduling for independent operations. However, if there are no opportunities for parallel execution, use the load/operation/store forms to reduce the number of register spills (storing values in memory to free registers for other uses).

Register Operands—Maintain frequently used values in registers rather than in memory.

Stack References—Use ESP for stack references so that EBP remains available.

Stack Allocation—When allocating space for local variables and/or outgoing parameters within a procedure, adjust the stack pointer and use moves rather than pushes. This method of allocation allows random access to the outgoing parameters so that they can be set up when they are calculated instead of being held somewhere else until the procedure call. This method also reduces ESP dependencies and uses fewer execution resources.

Data Embedding—When data is embedded in the code segment, align it in separate cache blocks from nearby code. This technique avoids some overhead when maintaining coherency between the instruction and data caches.

Loops—Unroll loops to get more parallelism and reduce loop overhead, even with branch prediction. Inline small routines to avoid procedure-call overhead. For both techniques, however, consider the cost of possible increased register usage, which might add load/store instructions for register spilling. Unrolling large code loops can result in the inefficient use of L1 instruction caches.

Code Alignment—Aligning subroutines at 0-mod-16 (or ideally, at 0-mod-32) address boundaries optimizes instruction cache-fill efficiency. Keeping the starting point of loops at least two instructions away from the end of 32-byte cache lines optimizes branch-target instruction fetch and decode efficiency.

General AMD-K6® Family x86 Coding Optimizations

This section describes general code optimization techniques specific to the AMD-K6 family of processors.

Use short-decodable instructions—To increase decode bandwidth and minimize the number of RISC86 operations per x86 instruction, use short-decodable x86 instructions. See "Instruction Dispatch" on page 27 for the list of short-decodable instructions.

Pair short-decodable instructions—Two short-decodable x86 instructions can be decoded per clock, using the full decode bandwidth of the AMD-K6 family.

Note: *For the AMD-K6-2 and AMD-K6-III processors, all MMX and 3DNow! instructions are short-decodable except the EMMS, FEMMS, and PREFETCH instructions.*

Avoid using complex instructions—The more complex and uncommon instructions are vector decoded and can generate a larger ratio of RISC86 operations per x86 instruction compared with short-decodable or long-decodable instructions.

Avoid multiple and accumulated prefixes—In order to accomplish an instruction decode, the decoders require sufficient predecode information. When an instruction has multiple prefixes and this cannot be deduced by the decoders (due to a lack of data in the instruction decode buffer), the first decoder retires and accumulates one prefix per cycle until the instruction is completely decoded. Table 10 shows when prefixes are accumulated and decoding is serialized.

Table 10.   Decode Accumulation and Serialization

Decode #1	Decoder #2	Results
Instruction		Single instruction decoded.
Instruction	Instruction	Dual instruction decode.
Instruction	Prefix	Single instruction decode and prefix is accumulated.
Prefix	Instruction (modified by Prefix)	No prefix accumulation and single instruction is decoded.

Table 10.   Decode Accumulation and Serialization (continued)

Decode #1	Decoder #2	Results
PrefixA	PrefixB	Accumulate PrefixA and cancel decode of the second prefix.
PrefixB	Instruction	If a prefix has already been accumulated in the previous decode cycle, accumulate PrefixB and cancel instruction decode, wait for next decode cycle to decode the instruction.

0Fh prefix usage—0Fh does not count as a prefix for the decoder accumulation rules (that is, it does not cause accumulation).

Avoid long instruction length—Use x86 instructions that are less than eight bytes in length. An x86 instruction that is longer than seven bytes cannot be short-decoded.

Use read-modify-write instructions over discrete equivalent—No advantage is gained by splitting read-modify-write instructions into a load-execute-store instruction group. Both read-modify-write instructions and load-execute-store instruction groups decode and execute in one cycle but read-modify-write instructions promote better code density.

Move rarely used code and data to separate pages—Placing code, such as exception handlers, in separate pages and data, such as error text messages, in separate pages maximizes the use of the TLBs and prevents table pollution with rarely used items.

Avoid mixing code size types—Size prefixes that affect the length of an instruction can sometimes inhibit dual decoding.

Always pair CALL and RETURN—If CALLs and RETs are not paired, the return address stack gets out of synchronization, increasing the latency of returns and decreasing performance.

Exploit parallel execution of integer and floating-point multiplies—The AMD-K6 processor allows simultaneous integer and floating-point multiplies using separate, low-latency multipliers.

Avoid more than 16 levels of nesting in subroutines—More than 16 levels of nested subroutine calls overflow the return address stack, leading to lower performance. While this is not a problem for most code, recursive subroutines might easily exceed 16 levels of subroutine calls. If the recursive subroutine is tail recursive, it can usually be mechanically transformed into an iterative version, which leads to increased performance.

Place frequently used stack data within 128 bytes of the EBP—The statically most-referenced data items in a function's stack frame should be located from –128 to +127 bytes from EBP. This technique improves code density by enabling the use of an 8-bit sign-extended displacement instead of a 32-bit displacement.

Avoid superset dependencies—Using the larger form of a register immediate after an instruction uses the smaller form creates a superset dependency and prevents parallel execution. For example, avoid the following type of code:

```
OR          AH,07h
ADD         EAX,1555555h
```

One method for avoiding superset dependencies is to schedule the instruction with the superset dependency (for example, the ADD instruction) 4–6 instructions later than would normally be preferable. Another method, useful in some cases, is to use the MOVZX instruction to efficiently convert a byte-size value to a doubleword-size value, which can then be combined with other values in 32-bit operations.

Avoid excessive loop unrolling or code inlining—Excessive loop unrolling or code inlining increases code size and reduces locality, which leads to lower cache hit rates and reduced performance.

Avoid splitting a 16-bit memory access in 32-bit code—No advantage is gained by splitting a 16-bit memory access in 32-bit code into two byte-sized accesses. This technique avoids the operand size override.

Avoid data dependent branches around a single instruction—Data dependent branches acting upon basically random data cause the branch prediction logic to mispredict the branch about 50% of the time. Design branch-free alternative code

sequences. The effect is shorter average execution time. The following examples illustrate this concept:

- Signed integer ABS function (x = labs(x))
  Static Latency: 4 cycles

  ```
  MOV   ECX, [x]   ;load value
  MOV   EBX, ECX
  SAR   ECX, 31
  XOR   EBX, ECX   ;1's complement if x<0, else don't modify
  SUB   EBX, ECX   ;2's complement if x<0, else don't modify
  MOV   [x], EBX   ;save labs result
  ```

  Unsigned integer min function (z = x < y ? x : y)
  Static Latency: 4 cycles

  ```
  MOV   EAX, [x]   ;load x value
  MOV   EBX, [y]   ;load y value
  SUB   EAX, EBX   ;set carry flag if y is greater than x
  SBB   ECX, ECX   ;get borrow out from previous SUB
  AND   ECX, EAX   ;if x > y, ECX = x-y, else 0
  ADD   ECX, EBX   ;if x > y, return x-y+y = x, else y
  MOV   [z], ECX   ;save min (x,y)
  ```

- Hexadecimal to ASCII conversion
  (y=x < 10 ? x + 0x30: x + 0x41)
  Static Latency: 4 cycles

  ```
  MOV   AL, [x]    ;load x value
  CMP   AL, 10     ;if x is less than 10, set carry flag
  SBB   AL, 69h    ;0..9 -> 96h, Ah..Fh -> A1h...A6h
  DAS              ;0..9: subtract 66h, Ah..Fh: Subtract 60h
  MOV   [y],AL     ;save conversion in y
  ```

Avoid using the [ESI] addressing mode— This addressing mode forces the instructions using it to become vector decoded. There are two ways to avoid this problem. The first way is to use another register. The second way is to alter the addressing mode by explicitly coding [ESI+0]. Assemblers may optimize this to [ESI] by removing the 0.

AMD-K6® Family Integer x86 Coding Optimizations

This section describes integer code optimization techniques specific to the AMD-K6 family of processors.

Neutral code filler—Use the XCHG EAX, EAX or NOP instruction when aligning instructions. XCHG EAX, EAX consumes one decode slot but requires no execution resources. Essentially, the scheduler absorbs the equivalent RISC86 operation without requiring any of the execution units.

Inline REP String with low counts—Expand REP String instructions into equivalent sequences of simple x86 instructions. This technique eliminates the setup overhead of these instructions and increases instruction throughput.

Use ADD reg, reg instead of SHL reg, 1—This optimization technique allows the scheduler to use either of the two integer adders rather than the single shifter and effectively increases overall throughput. The only difference between these two instructions is the setting of the AF flag.

Use MOVZX and MOVSX to zero-extend and sign-extend byte-size and word-size operands to doubleword length—For example, typical code for zero extension creates a superset dependency when the zero-extended value is used, as in the following code:

```
XOR         EAX,EAX
MOV         AL, [mem]
```

Instead, use the following code:

```
MOVZX       EAX,BYTE PTR [mem]
```

Use load-execute integer instructions—Most load-execute integer instructions are short-decodable and can be decoded at the rate of two per cycle. Splitting a load-execute instruction into two separate instructions—a load instruction and a reg, reg instruction—reduces decoding bandwidth and increases register pressure. The split-instruction form can be used to avoid scheduler stalls for longer executing instructions and to explicitly schedule the load and execute operations.

Use AL, AX, and EAX to improve code density—In many cases, instructions using AL and EAX can be encoded in one less byte than using a general-purpose register. For example, ADD AX, 0x5555 should be encoded 05 55 55 and not 81 C0 55 55.

Clear registers using MOV reg, 0 instead of XOR reg, reg—Executing XOR reg, reg requires additional overhead due to register dependency checking and flag generation. Using MOV reg, 0 produces a limm (load immediate) RISC86 operation that is completed when placed in the scheduler and does not consume execution resources.

Use 8-bit sign-extended immediates—Using 8-bit sign-extended immediates improves code density with no negative effects on the AMD-K6 processor. For example, ADD BX, –55 should be encoded 83 C3 FB and not 81 C3 FF FB.

Use 8-bit sign-extended displacements for conditional branches—Using short, 8-bit sign-extended displacements for conditional branches improves code density with no negative effects on the AMD-K6 processor.

Use integer multiply over shift-add sequences when it is advantageous—The AMD-K6 processor features a low-latency integer multiplier. Therefore, almost any shift-add sequences can have higher latency than MUL or IMUL instructions. An exception is a trivial case involving multiplication by powers of two by means of left shifts. In general, replacements should be made if the shift-add sequences have a latency greater than or equal to 3 clocks.

Carefully choose the best method for pushing memory data—To reduce register pressure and code dependency, use PUSH [mem] rather than MOV EAX, [mem] followed by PUSH EAX.

Balance the use of CWD, CBW, CDQ, and CWDE—These instructions require special attention to avoid either decreased decode or execution bandwidth. The following code illustrates the possible trade-offs:

■ The following code replacement trades decode bandwidth (CWD is vector decoded, but with only one RISC86 operation) with execution bandwidth (SAR requires two RISC86 operations, including a shift):

```
Replace:CWD       With: MOV    DX,AX
                        SAR    DX,15
```

■ The following code replacement improves decode bandwidth (CBW is vector decoded while MOVSX is short decoded):

```
Replace:CBW       With: MOVSX AX,AL
```

■ The following code replacement trades decode bandwidth (CDQ is vector decoded, but with only two RISC86 operations) with execution bandwidth (SAR requires two RISC86 operations, including a shifter):

```
Replace:CDQ       With: MOV   EDX,EAX
                        SAR   EDX,31
```

■ The following code replacement improves decode bandwidth (CWDE is vector decoded while MOVSX is short decoded):

```
Replace:CWDE      With: MOVSX EAX, AX
```

Replace integer division by constants with multiplication by the reciprocal—This optimization is commonly used on RISC processors. Because the AMD-K6 processor has an extremely fast integer multiply (two cycles) and the integer division delivers only two bits of quotient per cycle (approximately 18 cycles for 32-bit divides), the equivalent code is much faster. The following examples illustrate the use of integer division by constants:

■ Unsigned division by 10 using multiplication by reciprocal
Static Latency: 5 cycles

```
; IN: EAX = dividend
; OUT:EDX = quotient
MOV  EDX, 0CCCCCCCDh  ;0.1 * 2^32 * 8 rounded up
MUL  EDX
SHR  EDX, 3           ;divide by 2^32 * 8
```

■ Unsigned division by 3 using multiplication by reciprocal
Static Latency: 5 cycles

```
; IN: EAX = dividend
; OUT:EDX = quotient
MOV  EDX, 0AAAAAAABh  ;1/3 * 2^32 * 2 rounded up
MUL  EDX
SHR  EDX, 1           ;divide by 2^32 * 2
```

■ Signed division by 2
Static Latency: 3 cycles

```
; IN: EAX = dividend
; OUT:EAX = quotient
CMP  EAX, 800000000h  ;CY = 1, if dividend >=0
SBB  EAX, -1          ;increment dividend if it is <0
SAR  EAX, 1           ;perform a right shift
```

- **Signed division by 2^n**
  Static Latency: 5 cycles

```
; IN: EAX = dividend
; OUT:EAX = quotient
MOV  EDX, EAX          ;sign extend into EDX
SAR  EDX, 31           ;EDX = 0xFFFFFFFF if dividend < 0
AND  EDX, (2^n-1)      ;mask correction (use divisor -1)
ADD  EAX, EDX          ;apply correction if necessary
SAR  EAX, (n)          ;perform right shift by log2 (divisor)
```

- **Signed division by –2**
  Static Latency: 4 cycles

```
; IN: EAX = dividend
; OUT:EAX = quotient
CMP  EAX, 800000000h   ;CY = 1, if dividend >=0
SBB  EAX, -1           ;increment dividend if it is <0
SAR  EAX, 1            ;perform right shift
NEG  EAX               ;use (x/-2) = = - (x/2)
```

- **Signed division by –(2^n)**
  Static Latency: 6 cycles

```
; IN: EAX = dividend
; OUT:EAX = quotient
MOV  EDX, EAX          ;sign extend into EDX
SAR  EDX, 31           ;EDX = 0xFFFFFFFF if dividend < 0
AND  EDX, (2^n-1)      ;mask correction (-divisor -1)
ADD  EAX, EDX          ;apply correction if necessary
SAR  EAX, (n)          ;right shift by log2(-divisor)
NEG  EAX               ;use (x/-(2^n)) = = (- (x/2^n))
```

- **Remainder of signed integer 2 or (–2)**
  Static Latency: 4 cycles

```
; IN: EAX = dividend
; OUT:EDX = quotient
MOV  EDX, EAX          ;sign extend into EDX
SAR  EDX, 31           ;EDX = 0xFFFFFFFF if dividend < 0
AND  EDX, 1            ;compute remainder
XOR  EAX, EDX          ;negate remainder if
SUB  EAX, EDX          ;dividend was < 0
MOV  [quotient], EAX
```

- **Remainder of signed integer (2^n) or (–(2^n)))**
  Static Latency: 6 cycles

```
; IN: EAX = dividend
; OUT:EDX = quotient
MOV  EDX, EAX          ;sign extend into EDX
SAR  EDX, 31           ;EDX = 0xFFFFFFFF if dividend < 0
AND  EDX, (2^n-1)      ;mask correction (abs(divison)-1)
ADD  EAX, EDX          ;apply pre-correction
AND  EAX, (2^n-1)      ;mask out remainder (abs(divison)-1)
SUB  EAX, EDX          ;apply pre-correction if necessary
MOV  [quotient], EAX
```

AMD-K6®-2 and AMD-K6®-III Processors Multimedia Coding Optimizations

This section describes multimedia code optimization techniques for the AMD-K6-2 and AMD-K6-III processors.

For optimal floating-point performance—Wherever possible, use the packed single-precision, floating-point capability of 3DNow! technology instead of the single-precision, double-precision, and extended-precision floating-point capabilities of the x87 floating-point unit. The 3DNow! units are fully pipelined, allow vectorized optimizations, are not stack based, and provide faster inverse, square root, and inverse square root calculations.

Issues to ensure optimal predecode of MMX™ and 3DNow!™ instructions—Attention must be paid to coding issues that can inhibit the predecode, and later dual decode, of x86 instructions. Instructions are predecoded during instruction cache line fills. The predecode information that is produced and then stored in the predecode cache is later used by the instruction decoders to quickly find consecutive instructions and, therefore, enable dual-instruction decode. (The predecode information, in particular, reflects the length of instructions.)

The processor predecode scheme is based on a number of assumptions and constraints that have been mentioned previously, but which are repeated here for convenience:

- Only a subset of x86 instructions are short decodable and require predecode information. These include all MMX and 3DNow! instructions except for the EMMS, FEMMS, and PREFETCH instructions.
- Predecodable instructions can be up to seven bytes in length.
- The processor predecoders can only examine the first three bytes of an instruction to determine the length of the instruction and generate the predecode information. To determine instruction length, non-modR/M instructions require examination of the opcode byte, and modR/M instructions require the examination of the opcode byte plus the modR/M byte. Instructions with a 0Fh prefix

require the examination of the 0Fh byte in addition to the opcode byte and any modR/M byte. Finally, modR/M address modes with a sib byte and no displacement (modR/M = 00_xxx_100b) require examination of the additional sib byte. Instructions in this last category that also require a 0Fh prefix violate the three-byte predecode constraint and, therefore, cannot be predecoded—these instructions use [disp32 + index], [disp32 + scale • index], [base + index], or [base + scale • index] address modes and, therefore, require the examination of four bytes to determine instruction length. Note that the [base], [disp32], and [base + disp32] address modes are not affected by this.

■ The 32-bit modR/M address mode [ESI] cannot be predecoded.

■ For instructions starting within the last two bytes of a cache line, the predecode logic is not able to scan past the end of the cache line when it needs to examine more bytes to determine the length of an instruction. This constraint limits the type of instructions that can be predecoded at the end of a cache line. For example, a modR/M instruction that starts on the last byte of a 32-byte cache line, or a 0Fh-prefix plus modR/M instruction that starts within the last two bytes of the cache line, cannot be predecoded.

■ MMX and 3DNow! instructions have a 0Fh-prefix byte, an opcode byte, and a modR/M byte, all of which must be examined by the predecode logic.

These constraints result in the following recommendations for successful predecode of multimedia instructions:

■ With 3DNow! instructions, do not use address modes with large (32-bit) displacements. Large displacements result in a total instruction length of eight bytes (including the additional suffix byte used at the end of the instruction as a sub-opcode byte).

■ With MMX and 3DNow! instructions, do not use the [disp32 + index], [disp32 + scale • index], [base + index], [base + scale • index], or [ESI] address modes. Instead use the [base], [base + disp32], [disp32], or [ESI+0] (with byte offset) address modes.

■ Avoid placing the start of MMX and 3DNow! instructions in the last two bytes of a cache line. If not successfully predecoded, MMX instructions default to vector decodes and 3DNow! instructions default to long decodes.

A comparison of the instruction decode clock-cycle count on optimized code is as follows:

- 0.5 cycle for one short decode as part of a dual decode.
- 1.0 cycle for a single long decode.
- 2.0 cycles for a single vector decode (for simple instructions such as MMX and 3DNow! instructions).

Avoid using MMX™/3DNow!™ registers to move double-precision floating-point data—Although using an MMX/3DNow! register to move x87 floating-point data appears fast, using these registers requires the use of the EMMS or FEMMS instruction when switching from MMX or 3DNow! instructions to x87 instructions.

Use the FEMMS instruction instead of the EMMS instruction— The processor implements an improved version of the EMMS instruction, called FEMMS. Because the MMX/3DNow! registers are mapped onto the x87 stack, an EMMS or FEMMS instruction must be executed when switching from MMX or 3DNow! code to x87 code. Execution of the EMMS or FEMMS instruction marks the floating-point tag word as empty (all 1s), which guarantees correct x87 results and ensures that no x87 exceptions occur in the subsequent code due to a stack overflow.

Each time the processor encounters a switch between MMX or 3DNow! code and x87 code, in either direction, a significant clock-cycle count penalty occurs. The FEMMS instruction was created to reduce this penalty. The FEMMS instruction sets the floating-point tag word to empty (like EMMS), and also sets all of the register values as undefined. If a switch is required following a FEMMS instruction, it executes in less than half the cycles required after an EMMS instruction. The switch overhead occurs when an x87 instruction is encountered, and not during the execution of the EMMS and FEMMS instructions. In addition, the FEMMS instruction executes in 3 clock cycles, 2 cycles less than the EMMS instruction. For more information on the operation and advantages of the FEMMS instruction, see the *3DNow!™ Technology Manual,* order# 21928.

Use the FEMMS instruction at the beginning of an MMX™ or 3DNow!™ routine—While the FEMMS instruction is not necessary for correct program functionality at the beginning of MMX or 3DNow! routines, its usage reduces the clock-cycle count penalty when entering such routines from preceding x87 code. If no switch occurs, the FEMMS takes 3 clock cycles to execute. If a switch is necessary, FEMMS reduces the clock cycles required by over half.

Practice the following general rules when using MMX™ or 3DNow!™ code mixed with x87 code:

■ Always use the FEMMS instruction (instead of EMMS) at the end of an MMX or 3DNow! instruction routine when x87 instructions or unknown code follows.

■ Use the FEMMS instruction at the beginning of an MMX or 3DNow! instruction routine that is preceded by x87 instructions or unknown code. FEMMS serves to reduce any switch penalty.

■ Group or partition MMX or 3DNow! code separate from x87 code to minimize the frequency of switching between MMX or 3DNow! operations and x87 operations.

Use the new 3DNow!™ instruction PAVGUSB instruction for MPEG-2 motion compensation—In DVD decoding, motion compensation performs a lot of byte averaging between and within macroblocks. The PAVGUSB instruction helps speed up these operations. In addition, PAVGUSB can free up some registers and make unrolling the averaging loops possible.

The following code fragment uses original MMX code to perform averaging between the source macroblock and destination macroblock:

```
mov        esi, DWORD PTR Src_MB
mov        edi, DWORD PTR Dst_MB
mov        edx, DWORD PTR SrcStride
mov        ebx, DWORD PTR DstStride
movq       mm7, QWORD PTR [ConstFEFE]
movq       mm6, QWORD PTR [Const0101]
mov        ecx, 16

L1:
movq       mm0, [esi]      ;mm0=qword1
movq       mm1, [edi]      ;mm1=qword3
movq       mm2, mm0
movq       mm3, mm1
pand       mm2, mm6
pand       mm3, mm6
pand       mm0, mm7        ;mm0 = qword1 & 0xfefefefe
pand       mm1, mm7        ;mm1 = qword3 & 0xfefefefe
por        mm2, mm3        ;calculate adjustment
psrlq      mm0, 1          ;mm0 = (qword1 & 0xfefefefe)/2
psrlq      mm1, 1          ;mm1 = (qword3 & 0xfefefefe)/2
pand       mm2, mm6
paddb      mm0, mm1        ;mm0 = qw1/2 + qw3/2 w/o adjustment
paddb      mm0, mm2        ;add lsb adjustment
movq       [edi], mm0
movq       mm4, [esi+8]    ;mm4=qword2
movq       mm5, [edi+8]    ;mm5=qword4
movq       mm2, mm4
movq       mm3, mm5
pand       mm2, mm6
pand       mm3, mm6
pand       mm4, mm7        ;mm0 = qword2 & 0xfefefefe
pand       mm5, mm7        ;mm1 = qword4 & 0xfefefefe
por        mm2, mm3        ;calculate adjustment
psrlq      mm4, 1          ;mm0 = (qword2 & 0xfefefefe)/2
psrlq      mm5, 1          ;mm1 = (qword4 & 0xfefefefe)/2
pand       mm2, mm6
paddb      mm4, mm5        ;mm0 = qw2/2 + qw4/2 w/o adjustment
paddb      mm4, mm2        ;add lsb adjustment
movq       [edi+8], mm4

add        esi, edx
add        edi, ebx
loop       L1
```

The following code fragment uses the 3DNow! PAVGUSB instruction to perform averaging between the source macroblock and destination macroblock:

```
mov       eax, DWORD PTR Src_MB
mov       edi, DWORD PTR Dst_MB
mov       edx, DWORD PTR SrcStride
mov       ebx, DWORD PTR DstStride
mov       ecx, 16

L1:
movq      mm0, [eax]          ;mm0=qword1
movq      mm1, [eax+8]        ;mm1=qword2
pavgusb   mm0, [edi]          ;(qw1+qw3)/2 with adjustment
pavgusb   mm1, [edi+8]        ;(qw2+qw4)/2 with adjustment
add       eax, edx
movq      [edi], mm0
movq      [edi+8], mm1
add       edi, ebx
loop      L1
```

3DNow!™ Matrix Multiplication Optimization Example

The code samples starting on page 73 contain both a non-optimized and an optimized sample of a 4x4 matrix multiplied by a 4x1 vector. This type of code is often used in 3D graphics for geometry transformation. This routine serves to translate, scale, rotate, and apply perspective to 3D coordinates represented in homogeneous coordinates. The code samples contain many addition and multiplication instructions that can now be implemented in any one of three ways. For high-end, 3D graphic programs, x87 FPU instructions supply only moderate performance, are not superscalar, and cannot be efficiently intermixed with MMX and 3DNow! instructions. Integer instructions and MMX instructions, while fast and superscalar, do not have the accuracy and dynamic range that is required for these programs. Therefore, the 3DNow! instructions, providing the benefit of packed, floating-point data precision and parallel execution, can be used in order to write software that outperforms standard floating-point code and has no switching overhead when intermixed with MMX code. The following two code samples illustrate non-optimized and optimized code. A description of the steps a programmer should take when optimizing code for the AMD-K6-2 processor starts on page 78.

Non-Optimized Code Sample:

```
;-----------------------------------------------------------------------
; void Transform4x4(Vertex *firstv, int cnt, const Matrix *m)
;
; NON-OPTIMIZED VERSION
;
; Full 4x4 matrix transform of an array of cnt vertices starting from the
; vertex pointed to by firstv, using the transform matrix pointed to by m.
;
; Each vertex data structure is assumed to occupy 128 bytes, 16 bytes of
; which contains the vertex coordinates to be transformed.
;
;   new_x = x*m[0][0] + y*m[0][1] + z*m[0][2] + w*m[0][3];
;   new_y = x*m[1][0] + y*m[1][1] + z*m[1][2] + w*m[1][3];
;   new_z = x*m[2][0] + y*m[2][1] + z*m[2][2] + w*m[2][3];
;   new_w = x*m[3][0] + y*m[3][1] + z*m[3][2] + w*m[3][3];
;
;-----------------------------------------------------------------------
Vrtx_X equ  0h
Vrtx_Y equ  4h
Vrtx_Z equ  8h
Vrtx_W equ 0ch
Mat_00 equ  0h
Mat_01 equ  4h
Mat_02 equ  8h
Mat_03 equ 0ch
Mat_10 equ 10h
Mat_11 equ 14h
Mat_12 equ 18h
Mat_13 equ 1ch
Mat_20 equ 20h
Mat_21 equ 24h
Mat_22 equ 28h
Mat_23 equ 2ch
Mat_30 equ 30h
Mat_31 equ 34h
Mat_32 equ 38h
Mat_33 equ 3ch
                        ;EAX = m       ptr to transform matrix
                        ;EBX = firstv  ptr to first vertex to be transformed
                        ;EDX = lastv   ptr to last vertex to be transformed
```

Comments appear after the code lines.

```
TransformLoop:
                                    ;All multiplies for XResult:
    movq  mm0, QWORD PTR [ebx + Vrtx_X]  ;mm0 = y | x
    movq  mm2, mm0                       ;copy vector
```
Right in the beginning there is a dependency for mm0, which stalls the second movq 2 clock cycles, even though both instructions are short-decodable and decode together as an instruction pair.

```
pfmul mm0, QWORD PTR [eax + Mat_00]   ;mm0 = y*a21 | x*a11
```
The PFMUL instruction leads to another dependency, but because of the previous stall, the PFMUL instruction executes when Mat_00 loads from memory. The PFMUL instruction translates to a 3DNow! ALU and a Load unit operation.

```
movq  mm1, QWORD PTR [ebx + Vrtx_Z]   ;mm1 = w | z
```
The MOVQ instruction decodes with the previous PFMUL but there is now a resource constraint, with both instructions trying to use the Load unit. This contention causes one of the instructions to stall an extra cycle.

```
movq  mm3, mm1                          ;copy vector
```
Another stall while waiting for mm1.

```
pfmul mm1, QWORD PTR [eax + Mat_20]   ;mm1 = w*a41 | z*a31
```
Same as the previous PFMUL instruction. Note that tasks in this code line are serialized, with no opportunity for overlap of execution resources. Even if the instructions short decode in pairs, other constraints are causing stalls. In addition, a scheduler stall occurs when an instruction cannot retire off the bottom of the scheduler because dependency and resource stalls have delayed the instruction too many cycles.

```
                                        ;All multiplies for YResult:
movq  mm4, mm2                          ;copy vector
pfmul mm2, QWORD PTR [eax + Mat_01]   ;mm2 = y*a22 | x*a12
```
These instructions are paired. The PFMUL instructions decode to a Load unit operation followed by a 3DNow! Multiply unit operation.

```
movq  mm5, mm3                          ;copy vector
pfmul mm3, QWORD PTR [eax + Mat_21]   ;mm3 = w*a42 | z*a32
```
These instructions are paired. Same comments as before.

```
                                        ;All multiplies for ZResult:
movq  mm6, mm4                          ;copy vector
pfmul mm4, QWORD PTR [eax + Mat_02]   ;mm4 = y*a23 | x*a13
```
These instructions are paired. Same comments as before.

```
movq  mm7, mm5                          ;copy vector
pfmul mm5, QWORD PTR [eax + Mat_22]   ;mm5 = w*a43 | z*a33
```
These instructions are paired. Same comments as before.

```
                                        ;All multiplies for WResult:
pfmul mm6, QWORD PTR [eax + Mat_03]   ;mm6 = y*a24 | x*a14
pfmul mm7, QWORD PTR [eax + Mat_23]   ;mm7 = w*a44 | z*a34
```
These instructions are paired. However, this pair causes a conflict for both the Load unit and the 3DNow! Multiplier resources, which stalls one instruction in the scheduler for a clock cycle. The instructions execute in a staggered fashion. The goal for short-decodeable pairs is simultaneous execution.

```
                                        ;All first sums:
                                        ; of XResult
pfadd mm0, mm1                          ;mm0 = w*a41 + y*a21 | z*a31 + x*a11
                                        ; of YResult
pfadd mm2, mm3                          ;mm2 = w*a42 + y*a22 | z*a32 + x*a12
```
These instructions are paired. However, this pair causes a conflict for the 3DNow! ALU, which delay one instruction.

```
                                        ; of ZResult
pfadd mm4, mm5                          ;mm4 = w*a43 + y*a23 | z*a33 + x*a13
```

```
                                                ; of WResult
        pfadd mm6, mm7                          ;mm6 = w*a44 + y*a24 | z*a34 + x*a14
```
> *These instructions are paired, but there is a conflict for the 3DNow! ALU and with one of the PFADD instructions from the previous pair that was delayed one cycle. These dual-decodeable operations serialize execution, eventually stalling the scheduler because RISC86 instructions can no longer retire.*

```
                                        ;All final sums:
        pfacc mm0, mm0                  ; of XResult
        pfacc mm2, mm2                  ; of YResult
```
> *These instructions are paired, but there is a conflict for the 3DNow! ALU. See the comments above.*

```
        pfacc mm4, mm4                  ; of ZResult
        pfacc mm6, mm6                  ; of WResult
```
> *These instructions are paired, but there is a conflict for the 3DNow! ALU. See the comments above.*

```
                                        ;All result stores:
        movd  DWORD PTR [ebx + Vrtx_X], mm0  ; of XResult
        movd  DWORD PTR [ebx + Vrtx_Y], mm2  ; of YResult
```
> *These instructions are paired, but there is a conflict for the Store unit.*

```
        movd  DWORD PTR [ebx + Vrtx_Z], mm4  ; of ZResult
        movd  DWORD PTR [ebx + Vrtx_W], mm6  ; of WResult
```
> *These instructions are paired, but there is a conflict for the Store Unit as well as the delayed store operation from the previous instruction pair.*

```
        add   ebx, Vertex_Stride        ;Advance to next vertex
        cmp   ebx, edx                  ;Compare with ptr to last vertex
```
> *These instructions are paired, but a dependency on ebx value delays the second instruction by one cycle.*

```
        jbe   TransformLoop             ;If not done yet
```

Optimized Code Sample:

```
;----------------------------------------------------------------------------
; void Transform4x4(Vertex *firstv, int cnt, const Matrix *m)
;
; OPTIMIZED VERSION
;
; Full 4x4 matrix transform of an array of cnt vertices starting from the
; vertex pointed to by firstv, using the transform matrix pointed to by m.
;
; Each vertex data structure is assumed to occupy 128 bytes, 16 bytes of
; which contains the vertex coordinates to be transformed.
;
;   new_x = x*m[0][0] + y*m[0][1] + z*m[0][2] + w*m[0][3];
;   new_y = x*m[1][0] + y*m[1][1] + z*m[1][2] + w*m[1][3];
;   new_z = x*m[2][0] + y*m[2][1] + z*m[2][2] + w*m[2][3];
;   new_w = x*m[3][0] + y*m[3][1] + z*m[3][2] + w*m[3][3];
;
;----------------------------------------------------------------------------
Vrtx_X equ  0h
Vrtx_Y equ  4h
Vrtx_Z equ  8h
```

```
Vrtx_W equ 0ch
Mat_00 equ  0h
Mat_01 equ  4h
Mat_02 equ  8h
Mat_03 equ 0ch
Mat_10 equ 10h
Mat_11 equ 14h
Mat_12 equ 18h
Mat_13 equ 1ch
Mat_20 equ 20h
Mat_21 equ 24h
Mat_22 equ 28h
Mat_23 equ 2ch
Mat_30 equ 30h
Mat_31 equ 34h
Mat_32 equ 38h
Mat_33 equ 3ch
```

```
                         ;EAX = m       ptr to transform matrix
                         ;EBX = firstv  ptr to first vertex to be transformed
                         ;ECX = cnt     count of vertices to be transformed
```

The code begins here, but this section is not in the loop. The initial Loads conflict and stall waiting to load the first vertex values and the first four values from the matrix. However, once the loop begins, this code runs efficiently. Note that most of these x86 instructions are four bytes long, which helps to make them short decodable.

```
                                        ;Load first vertex:
movq  mm6, DWORD PTR [ebx]              ;mm6 = y | x
movq  mm7, DWORD PTR [ebx + Vrtx_Z]     ;mm7 = w | z
```

These instructions decode together, but cause a conflict for the Load unit.

```
                                        ;Start load of matrix:
movq  mm0, DWORD PTR [eax + Mat_00]     ;mm0 = m01 | m00
movq  mm1, DWORD PTR [eax + Mat_20]     ;mm1 = m03 | m02
```

Decode together, but conflict for the Load Unit.

```
TransformLoop:
    prefetchw [ebx + 128]                   ;Prefetch next vertex
```

The PREFETCHW instruction is a vector decode and takes 2 cycles. However, this instruction increases efficiency because it begins the preload of the L1 data cache with the next vertex. A vertex is four dwords or one half a cache line. However, the 'stride' or distance from one vertex data structure to the next within the vertex array, in this example, is 128 bytes, which means that each vertex is in a separate cache line. It is assumed that vertex data starts on cache line boundaries. From this point forward, the x86 instructions form instruction pairs that both decode into one Opquad. An Opquad is one line in the instruction scheduler that is composed of four RISC86 operations.

```
    movq  mm2, DWORD PTR [eax + Mat_01]        ;mm2 = m11 | m10
```

This MOVQ instruction continues to fill in the matrix. Separating the matrix load from the multiply instruction avoids serializing the load and multiply, which can lead to a stall of the scheduler. The load takes 2–3 cycles to execute and the multiply takes 2 cycles to execute. Including the operand fetch stage almost fills the six-stage length of the scheduler.

```
    pfmul mm0, mm6                             ;mm0 = y*m01 | x*m00
```

This PFMUL instruction is paired with the MOVQ instruction. These two instructions use different resources (Load Unit and 3DNow! ALU, respectively). There are no resource conflicts, no dependencies (mm0 should be loaded from three cycles earlier), and the instructions execute together.

```
movq  mm3, DWORD PTR [eax + Mat_21]      ;mm3 = m13 | m12
pfmul mm1, mm7                           ;mm1 = w*m03 | z*m02
```
Same comments as previous instruction pair.

```
movq  mm4, DWORD PTR [eax + Mat_02]      ;mm4 = m21 | m20
pfmul mm2, mm6                           ;mm2 = y*m11 | x*m10
```
Same comments as previous instruction pair, except the load mm2 was started two cycles earlier and should be forwarded from the Load unit to the 3DNow! ALU just-in-time.

```
movq  mm5, DWORD PTR [eax + Mat_22]      ;mm5 = m23 | m22
pfmul mm3, mm7                           ;mm3 = w*m13 | z*m12
```
In this pair of instructions, the last free register is loaded for now (mm5). Because there are only eight MMX registers, the registers must be reused and then reloaded with the matrix values for the next vertex calculation.

```
                                         ;First sum of XResult:
pfadd mm0, mm1                           ;mm0 = w*m03 + y*m01 | z*m02 + x*m00
pfmul mm4, mm6                           ;mm4 = y*m21 | x*m20
```
These two 3DNow! instructions can be paired because the 3DNow! ALU and multiplier are separate units and both have access to the issue buses for the register X and register Y execution pipelines. Note that at this time the processor is operating on eight single-precision, floating-point values (packed into four mmx registers) and the processor produces four single-precision values (in two mmx registers).

```
                                         ;First sum of YResult:
pfadd mm2, mm3                           ;mm2 = w*m13 + y*m11 | z*m12 + x*m10
```
The mm3 operand is forwarded from the 3DNow! multiplier output.

```
movq  mm1, DWORD PTR [eax + Mat_03]      ;mm1 = m31 | m30
```
The previous two instructions are paired. The MOVQ instruction moves in the first pair of the remaining four matrix values.

```
pfmul mm5, mm7                           ;mm5 = w*m23 | z*m22
```
The mm5 operand is forwarded from the Load unit and the mm7 operand is forwarded from the 3DNow! multiplier.

```
movq  mm3, DWORD PTR [eax + Mat_23]      ;mm3 = m33 | m32
```
The previous two instructions are paired. The MOVQ instruction moves in the last pair of matrix values.

```
add   ebx, Vertex_Stride                 ;Advance to next vertex
```
The pointer to the next vertex is updated. In this example, Vertex_Stride = 128.

```
pfmul mm1, mm6                           ;mm1 = y*m31 | x*m30
```
The previous two instructions are paired.

```
                                         ;Final sum of XResult and YResult:
pfacc mm0, mm2                           ;mm0 = YRes | XRes
```
The first pair of vertex values are complete and can be stored two clock cycles later (the 3DNow! accumulate instruction has a two-cycle execution latency, as do all 3DNow! ALU and Multiply instructions).

```
pfmul mm3, mm7                           ;mm3 = w*m33 | z*m32
```

The previous two instructions are paired and use the 3DNow! ALU and Multiplier units simultaneously.

```
                                         ;First sum of ZResult
pfadd mm4, mm5                           ;mm4 = w*m23 + y*m21 | z*m22 + x*m20
```

Continuing the goal of spreading out dependencies, this instruction is two cycles after the mm5 calculation.

```
                                         ;Load next vertex
movq  mm6, DWORD PTR [ebx + Vrtx_X]      ;mm6 = y | x
```

The previous two instructions are paired. This MOVQ instruction begins to load the next vertex, which the PREFETCHW instruction has been preloading into the L1 data cache.

```
                                         ;First sum of WResult:
pfadd mm1, mm3                           ;mm1 = w*m33 + y*m31 | z*m32 + x*m30
                                         ;Load next vertex
movq  mm7, DWORD PTR [ebx + Vrtx_Z]      ;mm7 = w | z
```

The previous two instructions are paired. The second part of the new vertex is loaded.

```
movq  DWORD PTR [ebx - 128 + Vrtx_X], mm0     ;Store XResult and YResult
                                              ;Start next iteration
movq  mm0, DWORD PTR [eax + Mat_00]   ;mm0 = m01 | m00
```

The previous two instructions are paired and can complete simultaneously because the AMD-K6-2 processor has separate Load and Store units. Unfortunately, all the matrix values must be reloaded with each iteration because there are not enough registers to hold the vertices, the full matrix, and intermediate values.

```
                                         ;Final sum of ZResult and WResult:
pfacc mm4, mm1                           ;mm4 = WRes | ZRes
                                         ;Start next iteration
movq  mm1, DWORD PTR [eax + Mat_20]   ;mm1 = m03 | m02
```

The previous two instructions are paired.

```
movq  DWORD PTR [ebx - 128 + Vrtx_Z], mm4     ;Store ZResult and WResult
```

Fortunately, the Store unit can accept data up to two cycles later without a penalty because there are no calculations left to hide the execution latency of the last accumulate instruction. Therefore, this store is not delayed.

```
loop  TransformLoop          ;If not done yet, go to beginning of the loop.
```

The previous two instructions are short decodeable and paired. Note that on the AMD-K6 family, the LOOP instruction executes in the same amount of time as the CMP and JBE instructions in the non-optimized example. However, the LOOP instruction, being only one instruction instead of two, is more efficient.

Programming Steps

The following descriptions review and expand on the steps taken to arrive at the optimized code example:

Schedule code into pairs of short-decodable x86 instructions that correspond to the expected decode pairing—Each short-decodable pair of instructions decodes into four RISC86 operations that form a set of four Op entries in the instruction scheduler. This set of entries moves down the scheduler and eventually retires from the bottom of the scheduler buffer. The scheduler buffer can hold a total of six sets of entries (which represents a total of 24 Op entries). Under ideal conditions of uninterrupted decode and execution (no stalls), these entries

also correspond to clock cycles through the scheduler and execution pipes of the AMD-K6-2 processor. Consequently, the programmer should schedule dependent instructions apart from each other, in different decode pairs, based on the execution latencies of the corresponding RISC86 operations. It is cleanest and simplest to use only MOVQ and MOVD instructions for memory loads and stores, and use register-to-register instructions for computations. In addition, this technique has the benefit of minimizing or avoiding instruction scheduling delays due to long-latency instructions (such as those with a memory load followed by a two-cycle register operation), not completing in time and, therefore, not being ready to commit results when the entry containing the associated RISC86 operations reaches the bottom of the scheduler buffer. This situation can lead to a stall when no new RISC86 operations can be placed in the scheduler until an entry is available.

Interleave independent sequences of instructions (subject to register allocation constraints) to fill each and every decode slot—To the extent that this is achieved while maintaining the proper minimum distances between dependent operations and respecting execution resource constraints, optimal decode pairing and instruction execution without delays or stalls is very likely to be achieved.

Use separate moves from memory and register-to-register multiplies, instead of register-to-register copies and multiplies from memory—This technique allows easy explicit and optimal scheduling of memory loads and dependent register operations, spaced at least two decode pairs apart and corresponding to the two-cycle load execution latency. While this technique generally applies to all MMX and 3DNow! instructions, particularly avoid the use of the memory form of instructions with two-cycle execution latencies (for example, all 3DNow! instructions). In other words, optimal performance is best and most easily achieved using a RISC coding style (despite the extra MOVD/MOVQ instructions).

Schedule instructions apart that use the same execution resources—For example, multiplies should be spread apart. The programmer should put at least one decode slot between multiplies. Similarly, adds and accumulates, memory loads, and memory stores should be spread apart.

Use the pipelining ability of accumulate instructions to perform two independent accumulates and to pair the resultant values together as a 64-bit result — This technique allows the use of fewer MOVQ instructions instead of a greater number of MOVD instructions. Overall, four accumulate and four MOVD instructions get replaced with two and two. In some situations, where scalar register results are naturally produced and are then stored out to memory via a series of MOVDs, it may be preferable to reduce the number of store operations through use of PUNPCKLDQ instructions followed by MOVQ instructions. Often this optimization may not be worthwhile or favorable, particularly given the extra latency introduced by the PUNPCKLDQ operations and possibly by memory alignment issues for the MOVQ instructions. Typically, it is best to spend the overhead to pack initial scalar operands together when first read from memory (using MOVD instructions), followed by vector computations and MOVQs back to memory.

Separate the first and second stores by at least two or three decode slots (in other words, by one intervening decode pair) within a series of two or more stores to a cache line recently brought in and not yet written to — This technique is in contrast to the second and following stores, which can be in adjacent decode pairs. This technique allows an extra cycle for the initial MESI-state change to the cache line (from Exclusive to Shared).

Schedule the ADD/CMP/JCC instructions apart (or at least the ADD and CMP instructions) — This scheduling is primarily desirable when the ADD and/or CMP instructions reference a memory operand and are, therefore, subject to the latency of the load operation. In such cases, either the ADD/CMP instruction should be scheduled apart from (and ahead of) the JCC instructions, or a separate MOV instruction, scheduled earlier, should be used to fetch the memory operand. An alternative and desirable solution in some cases is to replace these instructions with the LOOP instruction (along with corresponding setup and usage of the ECX register before and within the loop).

Take advantage of the PREFETCH instruction — In the optimized code example, each vertex occupies a different cache line (the stride between vertices being 32 bytes or greater). Consequently, one cache miss and associated 32 byte line fill occurs per loop iteration. To maximize overlap of the cache fill

from L2 cache or main memory, use the PREFETCH instruction to start the fill before starting to process the current vertex (which is already in the cache, having been prefetched at the beginning of processing of the last vertex). Specify the address elements of the next vertex that will be accessed first. In addition, schedule the loads of the next vertex's data elements away from the prefetch instruction. Doing so ensures that the load data (which will be the first data of the cache line to be fetched) has been received and is available for forwarding to these loads while the rest of the fill proceeds to completion.

Move the first few MOVQs around to the bottom of the loop— Typically, the first instructions after the prefetch instruction would be a series of MOVQs to get the first vertex and matrix elements to operate on, without any other available independent operations to fill out these first couple of decode pairs. Similarly, near the bottom of the loop, as the last computations are performed, there would also be some partially-filled decode slots. To fix both of these problems, move the first few vertex and matrix element MOVQ instructions from the bottom of the loop into the empty slots (as well as duplicating these MOVQs in the setup code before the start of the loop).

Pay attention to the alignment of instructions relative to 32-byte cache line boundaries—The code samples do not show the actual memory alignment of instructions and, therefore, whether the decode of any instructions may be impacted by end-of-cache-line degraded predecode. These code examples require a suitable starting alignment (relative to a 32-byte address boundary). There also exists the possibility that there is no starting alignment for which all instructions can be successfully predecoded. In such cases, adjustments to the code (such as padding with one-byte or multiple-byte NOPs, instruction rearrangement, or different instruction selections) may be warranted. In the case of 3DNow! instructions, which can still be hardware decoded as a single long decode, the best alternative may sometimes be to do nothing.

Avoid certain address modes with MMX™ and 3DNow!™ instructions that inhibit instruction predecode—As discussed earlier in the chapter, the [ESI] modR/M address mode (without any displacement bytes or index register) inhibits successful instruction predecode and should be avoided. In addition, for

MMX and 3DNow! instructions, address modes that use a sib byte with mod=00b in the modR/M byte should be avoided. These cases consist of [disp32 + index], [disp32 + scale • index], [base + index], or [base + scale • index] address modes.

Division and Square Root

Division
The 3DNow! instructions can be used to compute a very fast, highly accurate reciprocal or quotient.

Consider the quotient q = a/b. An (on-chip) ROM-based table lookup can be used to quickly produce a 14-to-15-bit precision approximation of 1/b. (Using just one 2-cycle latency PFRCP instruction). A full 24-bit precision reciprocal can then be quickly computed from this approximation using a Newton-Raphson algorithm.

The general Newton-Raphson recurrence for the reciprocal is as follows:

$$Z_{i+1} \leftarrow Z_i \bullet (2 - b \bullet Z_i)$$

Given that the initial approximation is accurate to at least 14 bits, and that full IEEE single precision contains 24 bits of mantissa, just one Newton-Raphson iteration is required. The following shows the 3DNow! instruction sequence to produce the initial reciprocal approximation, to compute the full-precision reciprocal from this, and lastly, to complete the required division of a/b.

$X_0 = \text{PFRCP}(b)$

$X_1 = \text{PFRCPIT1}(b, X_0)$

$X_2 = \text{PFRCPIT2}(X_1, X_0)$

$q = \text{PFMUL}(a, X_2)$

The 24-bit final reciprocal value is X_2. In the AMD processor implementation, the estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). The quotient is formed in the last step by multiplying the reciprocal by the dividend a.

Optimized 15-Bit Precision Divide

This divide operation executes with a total latency of 4 cycles, assuming that the programmer is able to hide the latency of the first MOVD/MOVQ instructions within preceding code.

```
MOVD       MM0, [mem]      ;   0|w
PFRCP      MM0, MM0        ;1/w|1/w
MOVQ       MM2, [mem]      ;   y|x
PFMUL      MM2, MM0        ;y/w|x/w
```

Optimized Full 24-Bit Precision Divide

This divide operation executes with a total latency of 8 cycles, assuming that the programmer is able to hide the latency of the first MOVD/MOVQ instructions within preceding code.

```
MOVD       MM0, [mem]      ;    0|w
PFRCP      MM1, MM0        ; 1/w|1/w
PUNPCKLDQ  MM0, MM0        ;    w|w
PFRCPIT1   MM0, MM1        ;
MOVQ       MM2, [mem]      ;    y|x
PFRCPIT2   MM0, MM1        ; 1/w|1/w
PFMUL      MM2, MM0        ; y/w|x/w
```

Pipelined Pair of 24-Bit Precision Divides

This divide operation executes with a total latency of 8 cycles, assuming that the programmer is able to hide the latency of the first MOVD/MOVQ instructions within preceding code.

```
MOVD       MM1, [mem]      ;  0  |w0
MOVD       MM2, [mem+4]    ;  0  |w1
PFRCP      MM1, MM1        ;1/w0|1/w0
MOVQ       MM0, [mem]      ;
PFRCP      MM2, MM2        ;1/w1|1/w1
PUNPCKLDQ  MM1, MM2        ;1/w1|1/w0
PFRCPIT1   MM0, MM1        ;
MOVQ       MM2, [mem]      ;    y|x
PFRCPIT2   MM0, MM1        ;1/w1|1/w0
PFMUL      MM2, MM0        ;y/w1|x/w0
```

Square Root and Reciprocal Square Root

The 3DNow! instructions can also be used to compute a reciprocal square root or square root with high performance. The general Newton-Raphson reciprocal square root recurrence is as follows:

$$Z_{i+1} \leftarrow 1/2 \bullet Z_i \bullet (3 - b \bullet Z_i^2)$$

To reduce the number of iterations, the initial approximation is read from a table. The 3DNow! reciprocal square root approximation is accurate to at least 15 bits. Accordingly, to obtain a single-precision 24-bit reciprocal square root of an

input operand b, one Newton-Raphson iteration is required using the following 3DNow! instructions:

1. X_0 = PFRSQRT(b)
2. X_1 = PFMUL(X_0, X_0)
3. X_2 = PFRSQIT1(b, X_1)
4. X_3 = PFRCPIT2(X_2, X_0)
5. X_4 = PFMUL(b, X_3)

The 24-bit final reciprocal square root value is X_3. In the AMD implementation, the estimate contains the correct round-to-nearest value for approximately 87% of all arguments. The remaining arguments differ from the correct round-to-nearest value for approximately 87% of all arguments. The remaining arguments differ from the correct round-to-nearest value by 1 ulp. The square root (X_4) is formed in the last step by multiplying by the input operand b.

Optimized 15-Bit Precision Square Root

This square root operation can be executed in only 4 cycles, assuming a programmer is able to hide the latency of the first MOVD instruction within previous code. The reciprocal square root operation requires two less cycles than the square root operation.

```
MOVD        MM0, [mem]      ;           0|a
PFRSQRT     MM1, MM0        ; 1/sqrt(a)|1/sqrt(a)
PUNPCKLDQ   MM0, MM0        ;           a|a
PFMUL       MM0, MM1        ;    sqrt(a)|sqrt(a)
```

Optimized 24-Bit Precision Square Root

This square root operation can be executed in only 10 cycles, assuming a programmer is able to hide the latency of the first MOVD instruction within previous code. The reciprocal square root operation requires two less cycles than the square root operation.

```
MOVD        MM0, [mem]      ;           0|a
PFRSQRT     MM1, MM0        ; 1/sqrt(a)|1/sqrt(a)
MOVQ        MM2, MM1        ;
PFMUL       MM1, MM1        ;
PUNPCKLDQ   MM0, MM0                   a|a
PFRSQIT1    MM1, MM0        ;
PFRCPIT2    MM1, MM2        ; 1/sqrt(a)|1/sqrt(a)
PFMUL       MM0, MM1        ;    sqrt(a)|sqrt(a)
```

AMD-K6®-2 and AMD-K6®-III Processors x87 Floating Point Coding Optimizations

This section describes x87 floating-point code optimization techniques specific to the AMD-K6-2 and AMD-K6-III processors.

For optimal floating-point performance—Wherever possible, use the packed single-precision, floating-point capability of 3DNow! technology instead of the single-precision, double-precision, and extended-precision floating-point capabilities of the x87 floating-point unit. The 3DNow! technology units are fully pipelined, allow vectorized optimizations, are not stack based, and provide faster inverse, square root, and inverse square root calculations.

Avoid vector decoded floating-point instructions—Most floating-point instructions are short decodable. A few of the less common instructions are vector decoded. In additional, if a short decodable instruction straddles a cache line, it becomes vector decoded. This adds unnecessary overheard that can be avoided by inserting NOPs in strategic locations within the code.

Pair floating-point with short-decodable instructions—Most floating-point instructions (also known as ESC instructions) are short-decodable and are limited to the first decoder. The short-decodable floating-point instructions can be paired with other short-decodable instructions. This technique requires that floating-point instructions be arranged as the first of a pair of short-decodable instructions.

Minimize switching between MMX™ or 3DNow!™ instructions and FPU instructions—Because the MMX/3DNow! registers are mapped onto the floating-point register stack, the EMMS or FEMMS instruction must be executed after MMX or 3DNow! code and prior to the use of the floating-point unit. Group or partition MMX and 3DNow! code away from FPU code so that the use of the EMMS or FEMMS instructions is minimized. In addition, the actual penalty or switch overhead from the use of the EMMS or FEMMS instructions occurs not at the time of their execution, but when and if the first floating-point instruction is encountered.

Avoid using MMX™/3DNow!™ registers (and MOVQ instructions) to move blocks of double-precision floating-point data in memory—Although using 64-bit MOVQ instructions to move floating-point data appears fast, using MMX/3DNow! registers requires the use of the EMMS or FEMMS instruction and incurs switch overhead when switching between these MMX or 3DNow! instructions and surrounding floating-point instructions.

Exploit parallel execution of integer and floating-point multiplies—The processor allows simultaneous integer and floating-point multiplies using separate, low-latency multipliers.

Do not split floating-point instructions with integer instructions—No penalty is incurred when using arithmetic or comparison floating-point instructions that use integer operands, such as the FIADD instruction or FICOM instruction. Splitting these instructions into discrete load and floating-point instructions decreases performance.

Replace FDIV instructions with FMUL where possible—The FMUL instruction latency is much less than the FDIV instruction. If possible, replace floating-point divisions with floating-point multiplication of the reciprocal.

Use integer instructions to move floating-point data—A floating-point load and store instruction pair requires a minimum of four cycles to complete (two-cycle latency for each instruction). The processor can perform one integer load and one store per cycle. Therefore, moving single-precision data requires one cycle, moving double-precision data requires two cycles, and moving extended-precision data only requires three cycles when using integer loads and stores. The following example shows how to translate the C-style code when moving double-precision floating-point data:

```
double temp1, temp2;
temp2 = temp1;

FLD   QWORD PTR [temp1];   Use: MOV  EAX, [temp1];
FSTP  QWORD PTR [temp2];        MOV  [temp2], EAX;
                                MOV  EAX, [temp1+4];
                                MOV  [temp2+4], EAX;
```

Scheduling of floating-point instructions is unnecessary—The processor has a low-latency, non-pipelined floating-point execution unit.

Use load-execute floating-point instructions—The use of a load-execute instruction (such as, FADD DWORD PRT [mem]) is preferable to the use of a load floating-point instruction followed by a floating-point reg, reg instruction. For the processor, load-execute arithmetic and compare instructions are identical in throughput to floating-point reg, reg instructions. Because common floating-point instructions execute in two cycles each and the floating-point unit is not pipelined, code executes more efficiently if the minimum possible number of floating-point instructions are generated.

Floating-Point Code Sample

The following code sample uses three of the most important rules to optimize this matrix multiply routine. The first rule used is avoidance of the [ESI] addressing mode. The routine forces this code to be [ESI+0]. The second rule is the insertion of NOPs to avoid cache-line straddles. The third rule used is avoidance of vector decoded instructions.

```
MATMUL    MACRO
   db     0d9h, 046h, 00h   ;; FLD DWORD PTR [ESI+00] ;;x
   FMUL   DWORD PTR [EBX]    ;; a11*x
   FLD    DWORD PTR [ESI+4] ;; y
   FMUL   DWORD PTR [EBX+4] ;; a21*y
   FLD    DWORD PTR [ESI+8] ;; z
   FMUL   DWORD PTR [EBX+8] ;; a31*z
   FLD    DWORD PTR [ESI+12];; w
   FMUL   DWORD PTR [EBX+12];; a41*w
   FADDP  ST(3), ST          ;; a41*w+a31*z
   FADDP  ST(2), ST          ;; a41*w+a31*z+a21*y
   FADDP  ST(1), ST          ;; a41*w+a31*z+a21*y+a11*x
   FSTP   DWORD PTR [EDI]    ;; store rx
   NOP                       ;; make sure it does not
                             ;; straddle across a cache line
   db     0d9h, 046h, 00h   ;; FLD DWORD PTR [ESI+00] ;; x
   FMUL   DWORD PTR [EBX+16];; a12*x
   FLD    DWORD PTR [ESI+4] ;; y
   FMUL   DWORD PTR [EBX+20];; a22*y
   FLD    DWORD PTR [ESI+8] ;; z
   NOP                       ;; make sure it does not
                             ;; straddle across a cache line
   FMUL   DWORD PTR [EBX+24];; a32*z
   FLD    DWORD PTR [ESI+12];; w
```

```
                FMUL  DWORD PTR [EBX+28];; a42*w
                FADDP ST(3), ST          ;; a42*w+a32*z
                FADDP ST(2), ST          ;; a42*w+a32*z+a22*y
                FADDP ST(1), ST          ;; a42*w+a32*z+a22*y+a12*x
                NOP                      ;; make sure it does not
                                         ;; straddle across a cache line
                FSTP  DWORD PTR [EDI+4]  ;; store ry
                db    0d9h, 046h, 00h    ;; FLD DWORD PTR [ESI+00] ;; x
                FMUL  DWORD PTR [EBX+32];; a13*x
                FLD   DWORD PTR [ESI+4]  ;; y
                FMUL  DWORD PTR [EBX+36];; a23*y
                NOP                      ;; make sure it does not
                                         ;; straddle across a cache line
                FLD   DWORD PTR [ESI+8]  ;; z
                FMUL  DWORD PTR [EBX+40];; a33*z
                FLD   DWORD PTR [ESI+12];; w
                FMUL  DWORD PTR [EBX+44];; a43*w
                FADDP ST(3), ST          ;; a43*w+a33*z
                FADDP ST(2), ST          ;; a43*w+a33*z+a23*y
                FADDP ST(1), ST          ;; a43*w+a33*z+a23*y+a13*x
                FSTP  DWORD PTR [EDI+8]  ;; store rz
                db    0d9h, 046h, 00h    ;; FLD DWORD PTR [ESI+00] ;; x
                FMUL  DWORD PTR [EBX+48];; a14*x
                FLD   DWORD PTR [ESI+4]  ;; y
                FMUL  DWORD PTR [EBX+52];; a24*y
                FLD   DWORD PTR [ESI+8]  ;; z
                FMUL  DWORD PTR [EBX+56];; a34*z
                FLD   DWORD PTR [ESI+12];; w
                FMUL  DWORD PTR [EBX+60];; a44*w
                FADDP ST(3), ST          ;; a44*w+a34*z
                NOP                      ;; make sure it does not
                                         ;; straddle across a cache line
                FADDP ST(2), ST          ;; a44*w+a34*z+a24*y
                FADDP ST(1), ST          ;; a44*w+a34*z+a24*y+a14*x
                FSTP  DWORD PTR [EDI+12];; store rw
            ENDM
```

6

Considerations for Other Processors

The tables in this chapter contain information describing how optimization techniques for the AMD-K6 family of processors affect other processors, including the AMD-K5 processor.

Table 11.   Specific Optimizations and Guidelines for AMD-K6® and AMD-K5™ Processors

AMD-K5™ Processor Guideline/Event	AMD-K5 Processor Details	Usage/Effect on AMD-K6® Processors	AMD-K6 Processor Details
Jumps and Loops	JCXZ requires 1 cycle (correctly predicted) and therefore is faster than a TEST/JZ. All forms of LOOP take 2 cycles (correctly predicted).	Different	JCXZ takes 2 cycles when taken and 7 cycles when not taken. LOOP takes 1 cycle.
Shifts	Although there is only one shifter, certain shifts can be done using other execution units. For example, shift left 1 by adding a value to itself. Use LEA index scaling to shift left by 1, 2, or 3.	Same	Shifts are short decodable and converted to a single RISC86 shift operation that executes only in the Integer X unit. LEA is executed in the store unit.
Multiplies	Independent IMULs can be pipelined at one per cycle with 4-cycle latency. (MUL has the same latency, although the implicit AX usage of MUL prevents independent, parallel MUL operations.)	Different	2– or 3–cycle throughput and latency. (3 cycles if the upper half of the product is produced.)

Table 11.   Specific Optimizations and Guidelines for AMD-K6® and AMD-K5™ Processors (continued)

AMD-K5™ Processor Guideline/Event	AMD-K5 Processor Details	Usage/Effect on AMD-K6® Processors	AMD-K6 Processor Details
Dispatch Conflicts	Load-balancing (that is, selecting instructions for parallel decode) is still important, but to a lesser extent than on the Pentium processor. In particular, arrange instructions to avoid execution-unit dispatching conflicts.	Same	
Byte Operations	For byte operations, the high and low bytes of AX, BX, CX, and DX are effectively independent registers that can be operated on in parallel. For example, reading AL does not have a dependency on an outstanding write to AH.	Same	Register dependency is checked on a byte boundary.
Floating-Point Top-of-Stack Bottleneck	The AMD-K5 processor has a pipelined floating-point unit. Greater parallelism can be achieved by using FXCH in parallel with floating-point operations to alleviate the top-of-stack bottleneck, as in the Pentium.	Not required	Loads and stores are performed in parallel with floating-point instructions.
Move and Convert	MOVZX, MOVSX, CBW, CWDE, CWD, CDQ all take 1 cycle (2 cycles for memory-based input).	Same	Zero and sign extension are short-decodable with 1 cycle execution.
Indexed Addressing	There is no penalty for base + index addressing in the AMD-K5 processor.	Same	
Instruction Prefixes	There is no penalty for instruction prefixes, including combinations such as segment-size and operand-size prefixes. This is particularly important for 16-bit code.	Possible	A penalty can only occur during accumulated prefix decoding.
Floating-Point Execution Parallelism	The AMD-K5 processor permits integer operations (ALU, branch, load/store) in parallel with floating-point operations.	Same	In addition, the AMD-K6 processor allows two integer, a branch, a load, and a store.
Locating Branch Targets	Performance can be sensitive to code alignment, especially in tight loops. Locating branch targets to the first 17 bytes of the 32-byte cache line maximizes the opportunity for parallel execution at the target.	Optional	Branch targets should be placed on 0 mod 16 alignment for optimal performance.

Table 11.   Specific Optimizations and Guidelines for AMD-K6® and AMD-K5™ Processors (continued)

AMD-K5™ Processor Guideline/Event	AMD-K5 Processor Details	Usage/Effect on AMD-K6® Processors	AMD-K6 Processor Details
NOPs	The AMD-K5 processor executes NOPs (opcode 90h) at the rate of two per cycle. Adding NOPs is even more effective if they execute in parallel with existing code.	Same	NOPs are short-decodable and consume decode bandwidth but no execution resources.
Branch Prediction	There are two branch prediction bits in a 32-byte instruction cache line. For effective branch prediction, code should be generated with one branch per 16-byte line half.	Not required	This optimization has a neutral effect on the AMD-K6 processor.
Bit Scan	BSF and BSR take 1 cycle (2 cycles for memory-based input), compared to the Pentium's data-dependent 6 to 34 cycles.	Different	A multi-cycle operation, but faster than Pentium.
Bit Test	BT, BTS, BTR, and BTC take 1 cycle for register-based operands, and 2 or 3 cycles for memory-based operands with immediate bit-offset. Register-based bit-offset forms on the AMD-K5 processor take 5 cycles.	Different	Bit test latencies are similar to the Pentium.

Table 12.   AMD-K6® Processor Versus Pentium® Processor-Specific Optimizations and Guidelines

Pentium® Guideline/Event	Pentium Effect	Usage/Effect on AMD-K6® Processors	AMD-K6 Processor Details
Instruction Fetches Across Two Cache Lines	No Penalty	Possible	Decode penalty only if there is not sufficient information to decode at least one instruction.
Mispredicted Conditional Branch Executed in U pipe	3-cycle penalty	Different	Mispredicted branches have a 1– to 4–cycle penalty.
Mispredicted Conditional Branch Executed in V pipe	4-cycle penalty	Different	Mispredicted branches have a 1– to 4–cycle penalty.
Mispredicted Calls	3-cycle penalty	None	
Mispredicted Unconditional Jumps	3-cycle penalty	None	
FXCH Optimizing	Pairs with most FP instructions and effectively hides FP stack manipulations.	None	

Table 12.   AMD-K6® Processor Versus Pentium® Processor-Specific Optimizations and Guidelines

Pentium® Guideline/Event	Pentium Effect	Usage/Effect on AMD-K6® Processors	AMD-K6 Processor Details
Index Versus Base Register	1-cycle penalty to calculate the effective address when an index register is used.	None	
Address Generation Interlock Due to Explicit Register Usage	1-clock penalty when instructions are not scheduled apart by at least one instruction.	None	However, it is best to schedule apart the dependency.
Address Generation Interlock Due to Implicit Register Usage	1-clock penalty when instructions are not scheduled apart by at least one instruction.	None	However, it is best to schedule apart the dependency.
Instructions with an Immediate Displacement	1-cycle penalty	None	
Carry & Borrow Instructions Issue Limits	Issued to U pipe only	Same	Issued to Integer X unit only.
Prefix Decode Penalty	1-clock delay	Possible	Delays can occur due to prefix accumulation.
0Fh Prefix Penalty	1-clock delay	None	
MOVZX Acceleration	No, incurs 4-cycle latency	Yes	Short-decodable, 1 cycle.
Unpairability Due to Register Dependencies	Incurred during flow and output dependency.	None	Dependencies do not affect instruction decode.
Shifts with Immediates Issue Limitations	Issued to U pipe only	Similar	Issued to the Integer X unit only.
Floating-Point Ops Issue Limitation	Issued to U pipe only	Similar	Issued to dedicated floating-point unit.
Conditional Code Pairing	Special pairing case	None	Conditional code such as JCCs are short decodable and pairable.
Integer Execution Delay Due to Transcendental Operation	Issue to U pipe is stalled	None	The AMD-K6 processor has a separate floating-point execution unit.
Instructions Greater Than 7 Bytes	Issued to U pipe only	Similar	Long and vector decodable only.
Misaligned Data Penalty	3-clock delay	Partial	1-clock delay.

Table 13.   AMD-K6® Processor and Pentium® Processor with Optimizations for MMX™ Instructions

Pentium®/MMX™ Guideline/Event	Pentium/MMX Effect	Usage/Effect on AMD-K6® Processor	AMD-K6 Processor Details
0Fh Prefix Penalty	None	None	
Three-clock Stalls for Dependent MMX Multiplies	Dependent instruction must be scheduled two instruction pairs following the multiply.	Different	Only two clock execution latency for all MMX multiples
Two-clock Stalls for Writing Then Storing an MMX Register	Requires scheduling the store two cycles after writing (updating) the MMX register.	None	
U Pipe: Integer/MMX Pairing	MMX instruction that access either memory or integer registers cannot be executed in the V pipe.	Different	No pairing restrictions.
U Pipe: MMX/Integer Pairing	V pipe integer instruction must be pairable.	Different	No pairing restrictions.
Pairing Two MMX Instructions	Cannot pair two MMX multiplies, two MMX shifts, or MMX instructions in V pipe with U pipe dependency.	None	No decode pairing restrictions. Optimum execution may still benefit, though, from such instructions not being paired together.
66h or 67h Prefix Penalty	Three clocks.	None	

Table 14.   AMD-K6® Processor and Pentium® Pro/Pentium II Specific Optimizations

Pentium® Pro/Pentium II Guideline/Event	Pentium Pro/Pentium II Effect	Usage/Effect on AMD-K6® Processor	AMD-K6 Processor Detail
Partial-Register Stalls	Avoid reading a large register after writing a smaller version of the same register. This causes the processor to stall the issuing of instructions that reference the full register and all subsequent instructions until after the partial write has retired. If the partial register update is adjacent to a subsequent full register read, the stall lasts at least seven clock cycles with respect to the decoder outputs. On the average, such a stall can prevent from 3 to 21 micro-ops from being issued.	Partial	The AMD-K6 processor performs register dependency checking on a byte granularity. Due to shorter pipelines, execution latency, and commitment latency, instruction issuing is not affected. However, execution is stalled.

Table 14. AMD-K6® Processor and Pentium® Pro/Pentium II Specific Optimizations (continued)

Pentium® Pro/Pentium II Guideline/Event	Pentium Pro/Pentium II Effect	Usage/Effect on AMD-K6® Processor	AMD-K6 Processor Detail
Branches	Exploit the processor return stack by using a RET rather than a JMP at the end of a subroutine.	Same	The AMD-K6 processor contains a Call/Return stack.
Avoid Self-Modifying Code	Code that alters itself can cause the processor to flush the processor's pipelines and can invalidate code resident in caches.	Same	
Code Alignment	16-byte block	Same	
Predicted Branch Penalty	BTB suffers 1-cycle delay	None	The AMD-K6 processor uses parallel adders for on-the-fly address generation.
Mispredicted Branch	Minimum 9, typically 10 to 15 clocks	Different	1 to 4 clocks.
Misaligned Data Penalty	More than 3 clocks	Partial	1-clock delay.
2-Byte Data Alignment	4-byte boundary	Same	The misalignment penalty is only a 1-clock delay.
4-Byte Data Alignment	4-byte boundary	Same	The misalignment penalty is only a 1-clock delay.
8-Byte Data Alignment	8-byte boundary	Same	The misalignment penalty is only a 1-clock delay.
Instruction Lengths Greater Than 7 Bytes	Issued one at a time or vectored	Different	Long-decodable and vector-decodable.
Prefix Penalty	1-clock delay	Possible	Delays can sometimes occur due to prefix accumulation.
0Fh Prefix Penalty	None	None	
MOVZX Acceleration	Yes	Yes	Short-decodable, 1 cycle.
Static Prediction Penalty	6 clocks	Different	3 clocks.

Table 15.   AMD-K6® Processor and Pentium® Pro with Optimizations for MMX™ Instructions

Pentium® Pro/Pentium II Guideline/Event	Pentium Pro/Pentium II Effect	Usage/Effect on AMD-K6® Processor	AMD-K6 Processor Details
Three Clock Stalls for Dependent MMX™ Multiplies	Dependent instruction must be scheduled two instruction pairs following the multiply.	None	Only two clock execution latency for all MMX multiples.
Pairing Two MMX Instructions	Cannot pair two MMX multiplies, or two MMX shifts.	Same	
Predicted Branches not in the BTB	~5-cycle latency	Different	1-cycle latency for BTB miss.