AMD

AMD-K6™ MMX™ Enhanced Processor

x86 Code Optimization

Application Note

Publication # **21828** Rev: **A** Amendment/**0**
Issue Date: **August 1997**

Contents

List of Tables

List of Figures

Revision History

Date	Rev	Description
August 1997	A	Initial Release

1

Introduction

Purpose

The AMD-K6™ MMX™ enhanced processor is the newest microprocessor in the AMD K86™ family of microprocessors. The AMD-K6 processor can efficiently execute code written for previous-generation x86 processors. However, there are many ways to get higher performance from the AMD-K6 processor.

This document contains information to assist programmers in creating optimized code for the AMD-K6 processor. 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 microarchitecture.

The AMD-K6™ Family of Processors

Processors in the AMD-K6™ family use a decoupled instruction decode and execution superscalar microarchitecture, including state-of-the-art RISC design techniques, to deliver sixth-generation performance with full 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.

The AMD-K6™ MMX™ Enhanced Processor

The AMD-K6 MMX enhanced processor, the first in the AMD-K6 family, brings superscalar RISC performance to desktop systems running industry-standard x86 software. This 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, and dynamic branch prediction. In other words, the AMD-K6 processor is capable of issuing, executing, and retiring multiple x86 instructions per cycle, resulting in superior scaleable performance.

Although the AMD-K6 processor is capable of extracting code parallelism out of off-the-shelf, commercially available x86 software, specific code optimizations for the AMD-K6 processor can result in even higher delivered performance. This document describes the RISC86® microarchitecture in the AMD-K6 processor and makes recommendations for optimizing execution of x86 software on the processor. The coding techniques for achieving peak performance on the AMD-K6 processor include, but are not limited to, those recommended for the Pentium® and Pentium Pro processors. However, many of these optimizations are not necessary for the AMD-K6 processor to achieve maximum performance. Due to the more flexible pipeline control of the AMD-K6 microarchitecture, the AMD-K6 processor is not as sensitive to instruction selection and the scheduling of code. This flexibility is one of the distinct advantages of the AMD-K6 processor microarchitecture.

2

The AMD-K6™ Processor 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 MMX 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, second-order 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.

RISC86® Microarchitecture

The enhanced RISC86 microarchitecture defines the characteristics of the AMD-K6 MMX enhanced processor. 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 enables higher processor core performance and promotes straightforward extendibility in future designs. Instead of executing complex x86 instructions, which have lengths of 1 to 15 bytes, the AMD-K6 processor executes the simpler fixed-length RISC86 opcodes, while maintaining the instruction coding efficiencies found in x86 programs.

The AMD-K6 processor includes parallel decoders, a centralized scheduler, and seven execution units that support superscalar operation—multiple decode, execution, and retirement—of x86 instructions. These elements are packed into an aggressive and very efficient six-stage 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, alongside the x86 instructions, in a dedicated level-one predecode cache to be used later by the decoders. Up to two x86 instructions are decoded per clock on-the-fly, resulting in a maximum of four RISC86 operations per clock with no additional latency.

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 vectored decode at a time. The three types of decodes have the following characteristics:

- Short decode—common x86 instructions less than or equal to 7 bytes in length that produce one or two RISC86 operations.

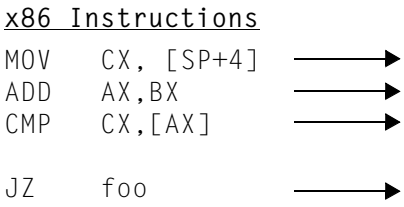
- Long decode—more complex and somewhat common x86 instructions less than or equal to 11 bytes in length that produce up to four RISC86 operations.
- Vectored decode—complex x86 instructions requiring long sequences of RISC86 operations.

Short and long decodes are processed completely within the decoders. Vectored decodes are started by the decoders 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 sent to a buffer in the centralized scheduler for dispatch to the execution units.

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

- Memory load operations (load)
- Memory store operations (store)
- Integer register operations (alu/alux)
- MMX register operations (meu)
- Floating-point register operations (float)
- Branch condition evaluations (branch)

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



The MOV instruction converts to a RISC86 load that requires indirect data to be loaded from memory. The ADD instruction converts to an alu function that can be sent to either of the integer units. The CMP instruction converts into two RISC86 instructions. 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 in the centralized scheduler buffer they are ready for the scheduler to issue them to the appropriate execution unit. The AMD-K6 processor contains seven execution units—Integer X, Integer Y, Multimedia, Load, Store, Branch, and Floating-Point. Figure 1 shows a block diagram of these units.

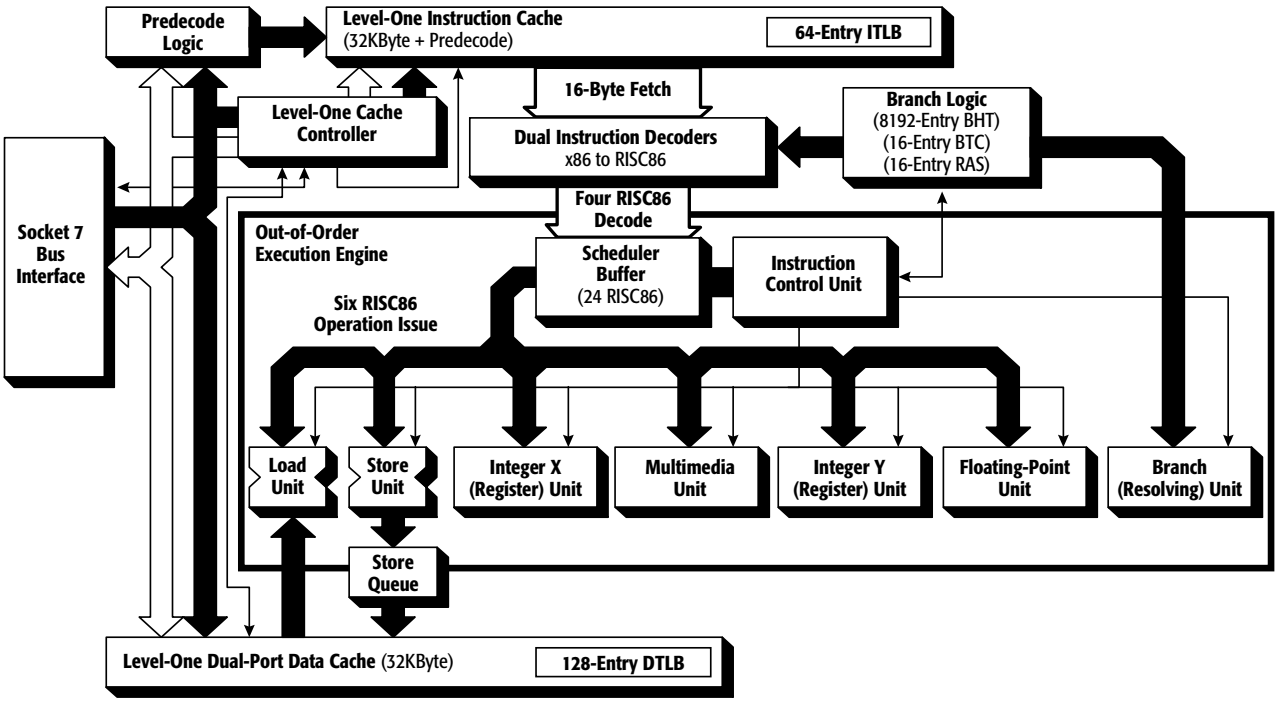


Figure 1. AMD-K6™ MMX™ Enhanced Processor Block Diagram

The centralized scheduler buffer, in conjunction with the instruction control unit (ICU/scheduler), buffers and manages up to 24 RISC86 operations at a time (which equates with up to 12 x86 instructions). This buffer size (24) is well matched to the processor's six-stage RISC86 pipeline and seven parallel execution units.

On every clock, the centralized scheduler buffer can accept up to four RISC86 operations from the decoders and issue up to six RISC86 operations to corresponding execution units. (Six RISC86 operations can be issued at a time because the alux and multimedia execution units share the same pipeline.)

When managing the 24 RISC86 operations, the scheduler uses 48 physical registers contained within the RISC86

microarchitecture. The 48 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 eight registers that correspond to the x86 general-purpose registers—EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI.

The AMD-K6 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 AMD-K6 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/RET 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 RET instruction.

As shown in Figure 1 on page 6, the high-performance, out-of-order execution engine is mated to a split 64-Kbyte (Harvard architecture) writeback level-one cache 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 an industry-standard 64-bit Pentium processor-compatible demultiplexed address/data system bus.

The AMD-K6 processor uses the latest in processor microarchitecture techniques to provide the highest x86 performance for today's PC. In short, the AMD-K6 processor offers true sixth-generation performance and full x86 binary software compatibility.

3

AMD-K6™ Processor Execution Units and Dependency Latencies

The AMD-K6 MMX enhanced processor contains seven specialized execution units—store, load, integer X, integer Y, multimedia, floating-point, and branch condition. Each unit operates independently and handles a specific group of the RISC86 instruction set. This chapter describes the operation of these units, their execution latencies, and how concurrent dependency chains affect those latencies.

A dependency occurs when data needed in one execution unit is being processed in another unit (or the same unit). Additional latencies can occur because the dependent execution unit must wait for the data. Table 1 on page 16 provides a summary of the execution units, the operations performed within these units, the operation latency, and the operation throughput.

Execution Unit Terminology

The execution units operate on two different types of register values—operands and results. 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*—from load or register operations
- *Address register results*—from 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.

Six-Stage Pipeline

To help visualize the operations within the AMD-K6 processor, Figure 2 illustrates the six-stage pipeline design. This is a simplified illustration in that the AMD-K6 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 AMD-K6 execution pipeline illustrates the effect of execution latencies for various types of operations.

For register operations that only require one execution cycle, this pipeline is effectively shorter due to the absence of execution stage 2.

The samples starting on page 19 assume that the x86 instructions have already been fetched, decoded, and placed in the centralized scheduler buffer. The RISC86 operations are waiting to be dispatched to the appropriate execution units.

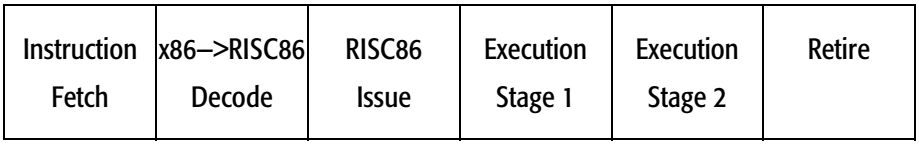


Figure 2. AMD-K6™ Processor Pipeline

Integer and Multimedia Execution Units

The integer X execution unit can execute all ALU operations, multiplies and divides (signed and unsigned), shifts, and rotates. Data register results are available after one clock of execution latency.

The multimedia execution unit (meu) executes all MMX operations and shares pipeline control with the integer X execution unit (an integer X operation and an MMX operation cannot be dispatched simultaneously). In most cases, data register results are available after one clock and after two clocks for PMULH and PMADD operations.

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

Figure 3 shows the architecture of the single-stage integer execution pipeline. The operation issue and fetch stages that precede this execution stage are not part of the execution pipeline. The data 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 execution pipe stage.

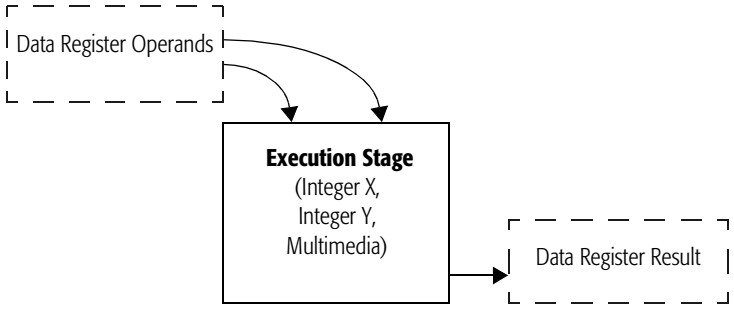


Figure 3. Integer/Multimedia Execution Unit

Load Unit

The load unit is a two-stage pipelined design that performs data memory reads. This unit uses two address register operands and a memory data value as inputs and produces a data register result.

The load unit has a two-clock latency from the time it receives the address register operands until it produces a data register result.

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

Figure 4 shows the architecture of the two-stage load 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 data register result is produced near the end of the second execution pipe stage.

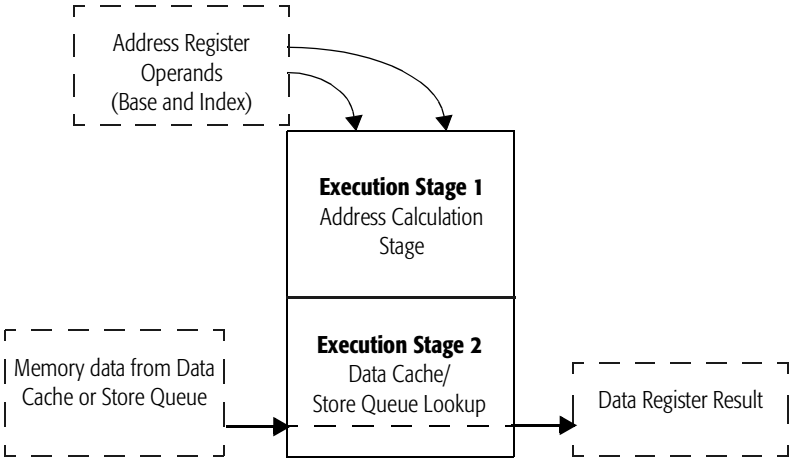


Figure 4. Load Execution Unit

Store Unit

The store execution unit is a two-stage pipelined design that performs data memory writes and/or, 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, which actually write to memory, the store unit produces a physical memory address and the associated bytes of data to be written. After execution completes, these results are entered in a new store queue entry.

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 both an address register result and performs a memory write.

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

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

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

Figure 5 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.

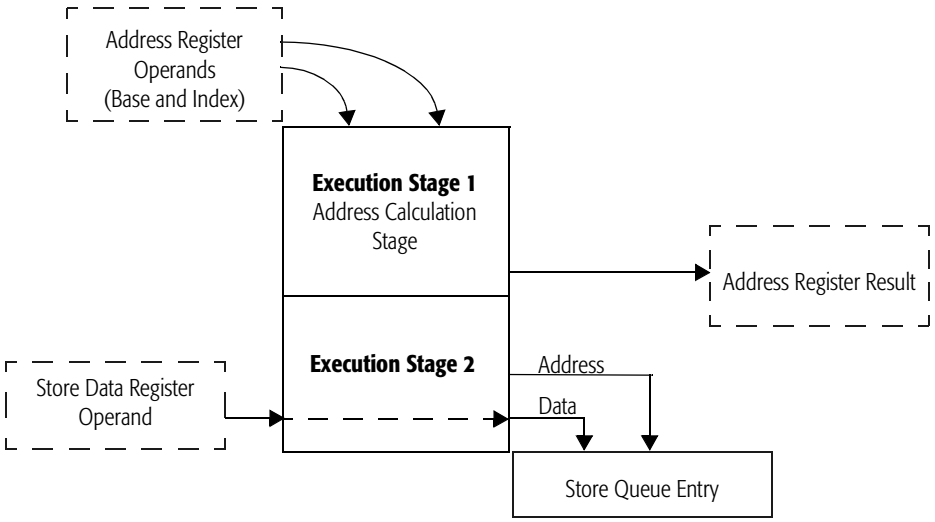


Figure 5. Store Execution Unit

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.

Floating Point Unit

The floating-point unit 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 it receives data register operands until the time it produces a data register result.

Latencies and Throughput

Table 1 on page 16 summarizes the latencies and throughput of each execution unit.

Table 1. RISC86® Execution Latencies and Throughput

Execution Unit	Operations	Latency	Throughput
Integer X	Integer ALU	1	1
	Integer Multiply	2–3	2–3
	Integer Shift	1	1
Multimedia	MMX ALU	1	1
	MMX Shifts, Packs, Unpack	1	1
	MMX Multiply Low/High	1/2	1/2
	MMX Multiply-Accumulate	2	2
Integer Y	Basic ALU (16– and 32– bit operands)	1	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 Operands 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, consider not only the latencies of critical dependencies, but also execution resource constraints. Due to a fixed number of execution units, only so many operations can be issued in each cycle (up to 6 RISC86 operations per cycle), even though, based on dependencies, more execution parallelism may be possible.

For example, if code contains three consecutive integer operations that do not have co-dependencies, they cannot execute in parallel because there are only 2 integer execution units. The third operation is delayed by one cycle.

Contention for execution resources causes delays in the issuing and execution of instructions. In addition, stalls due to resource constraints can combine with 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, the clock counts, and a description of the events occurring within the processor.

The following nomenclature is used to describe the current location of a RISC86 operation (RISC86op):

- D — Decode stage
- I_X — Issue stage of integer X unit
- O_X — Operand fetch stage of integer X unit
- E_{X1} — Execution stage 1 of integer X unit
- I_Y — Issue stage of integer Y unit
- O_Y — Operand fetch stage of integer Y unit
- E_{Y1} — Execution stage 1 of integer 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	RISC86op	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 AX and DX, 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 re-issued in the next cycle to the integer X unit (just in time for availability of its operands)

6 This add alu falls through to the integer Y unit right behind the first issuance of operation #5 without delay (as a result of operation #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	RISC86op	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	O_X	O_X	E_{X1}

Comments for Each Instruction Number

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

2 This operation will occupy 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 re-issued 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 receding instruction #3.

5 The register operand for the load operation is bypassed, without delay, from the result of instruction #2's register operand. Note how this and most surrounding load operations are generated by instruction decoders, and issued and executed by the load unit "smoothly" 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 instructions #3 and #4.

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

Instruction Number	Instruction	RISC86op	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	E_{L1}	E_{L2}					
		alu			I_X	O_X	I_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	O_Y	O_Y	O_Y	O_Y	E_{Y1}

Comments for Each Instruction Number

1 This operation will occupy 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 BX result in instruction #5.

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

Instruction Number	Instruction		RISC86op	Clocks									
				1	2	3	4	5	6	7	8	9	10
1	MOVQ	MM0, [EAX]	mload	D	I_L	O_L	E_{L1}	E_{L2}					
2	PSUBSW	MM0, [EAX+16]	mload		D	I_L	O_L	E_{L1}	E_{L2}				
			alux			I_X	O_X	O_X	O_X	E_{X1}			
3	ADD	EBX, ECX	alu		D	I_Y	O_Y	E_{Y1}					
4	PADDSW	MM1, MM2	alux			D	I_X	I_X	I_X	O_X	E_{X1}		
5	PUSH	EBX	store			D	I_S	O_S	E_{S1}	E_{S2}			
6	PMADDWD	MM0, MM1	alux				D			I_X	O_X	E_{X1}	E_{X1}
7	ADD	EAX, 32	alu				D	I_Y	O_Y	E_{Y1}			
8	MOVQ	[EDI], MM0	mstore					D	I_S	O_S	E_{S1}	E_{S2}	E_{S2}
9	ADD	EDI, 8	alu					D	I_Y	O_Y	E_{Y1}		

Comments for Each Instruction Number

1 This multimedia operation will occupy the load unit.

2 Instruction #2 could not be decoded along with the preceding instruction because only MMX instructions can be decoded in the first decode position. The MMX register operation is executable only by the integer X unit. The operand fetch is delayed because of the dependency of the load.

3 This instruction can be decoded in parallel with instruction #2 because it is not an MMX instruction. It is issued to the integer Y unit in parallel with the issuing of the preceding MMX register operation in instruction #2.

4 This instruction is only executable in the integer X unit. The issue of this MMX instruction is delayed due to the delay of the operand fetch of the preceding MMX register operation.

5 This instruction stores the contents of BX in memory.

6 Instruction #6 is only executable in the integer X unit. This non-pipelined unit has a two-clock execution latency for this instruction, and it is delayed due to 'stacking-up' behind the preceding MMX operations.

7 This instruction is issued to the integer Y unit in parallel with the series of preceding MMX register operations being issued to the integer X unit.

8 Completion of this store operation is held up due to a data dependency on the preceding MMX register operation from instruction #6. The store data is bypassed directly into a new store queue entry from the result of the MMX operation.

9 This instruction is issued to the integer Y unit in parallel with the series of preceding MMX register operations being issued to the integer X unit.

4

Instruction Dispatch and Execution Timing

This chapter describes the RISC86 operations executed by each instruction. Three separate tables define the integer, MMX, and floating-point instructions.

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 defined by bits 2, 1, and 0 of the modR/M byte
- *reg16/32*—word and doubleword integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte
- *mreg16/32*—word and doubleword integer register defined by bits 2, 1, and 0 of the modR/M byte
- *mem8*—byte memory location
- *mem16/32*—word or doubleword memory location
- *mem32/48*—doubleword or 6-byte memory location
- *mem48*—48-bit integer value in memory
- *mem64*—64-bit value in memory
- *imm8/16/32*—8-bit, 16-bit or 32-bit immediate value
- *disp8*—8-bit displacement value
- *disp16/32*—16-bit or 32-bit displacement value
- *disp32/48*—32-bit 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 register
- *mmreg1*—MMX register defined by bits 5, 4, and 3 of the modR/M byte
- *mmreg2*—MMX 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 vectored. The AMD-K6 MMX enhanced processor decode logic can process two short, one long, or one vectored decode per clock. In addition, two short integer, one short integer and one short MMX, or one short integer and one short FPU instruction can be decoded simultaneously.

Note: *In order to simultaneously decode an integer with a floating-point or MMX instruction, the floating-point or MMX instruction must precede the integer instruction.*

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 execution units
- *alux*—integer X execution unit only
- *branch*—branch condition unit
- *float*—floating-point execution unit
- *meu*—multimedia execution unit
- *limm*—load immediate, instruction control unit

The operation(s) of most instructions form a single dependency chain. For instructions whose operations form two parallel dependency chains, the RISC86 operations and execution latency 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® Opcodes
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		xx-xxx-xxx	vector	
ADC EAX, imm16/32	15h		xx-xxx-xxx	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		xx-xxx-xxx	short	alux

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
ADD EAX, imm16/32	05h		xx-xxx-xxx	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		xx-xxx-xxx	short	alux
AND EAX, imm16/32	25h		xx-xxx-xxx	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		xx-xxx-xxx	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® Opcodes
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® Opcodes
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		xx-xxx-xxx	short	alux
CMP EAX, imm16/32	3Dh		xx-xxx-xxx	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	long	load, alu
CMP mreg16/32, imm8 (signed ext.)	83h		11-111-xxx	short	load, alu
CMP mem16/32, imm8 (signed ext.)	83h		mm-111-xxx	short	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	
CMPXCH8B EDX:EAX	0Fh	C7h	11-xxx-xxx	vector	
CMPXCH8B 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® Opcodes
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-xx	vector	
DIV EAX, mreg16/32	F7h		11-110-xxx	vector	
DIV EAX, mem16/32	F7h		mm-110-xx	vector	
IDIV mreg8	F6h		11-111-xxx	vector	
IDIV mem8	F6h		mm-111-xx	vector	
IDIV EAX, mreg16/32	F7h		11-111-xxx	vector	
IDIV EAX, mem16/32	F7h		mm-111-xx	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-xx	vector	
IMUL EDX:EAX, EAX, mreg16/32	F7h		11-101-xxx	vector	
IMUL EDX:EAX, EAX, mem16/32	F7h		mm-101-xx	vector	
IMUL reg16/32, mreg16/32	0Fh	AFh	11-xxx-xxx	vector	
IMUL reg16/32, mem16/32	0Fh	AFh	mm-xxx-xxx	vector	
INC EAX	40h			short	alu
INC ECX	41h			short	alu
INC EDX	42h			short	alu
INC EBX	43h			short	alu

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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
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

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	
JMP far disp32/48 (direct)	EAh			vector	
JMP disp8 (short)	EBh			short	
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	alu store
LEAVE	C9h			long	alu load, 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	
LLDT mem16	0Fh	00h	mm-010-xxx	vector	
LMSW mreg16	0Fh	01h	11-100-xxx	vector	

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
LMSW mem16	0Fh	01h	mm-100-xxx	vector	
LODSB AL, mem8	ACh			long	load alu
LODSW AX, mem16	ADh			long	load alu
LODSD EAX, mem32	ADh			long	load alu
LOOP disp8	E2h			short	branch alu
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

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
MOV DL, imm8	B2h			short	limm
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 alu alu
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

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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-xx	vector	
MUL EAX, mreg16/32	F7h		11-100-xxx	vector	
MUL EAX, mem16/32	F7h		mm-100-xx	vector	
NEG mreg8	F6h		11-011-xxx	short	alux
NEG mem8	F6h		mm-011-xx	vector	
NEG mreg16/32	F7h		11-011-xxx	short	alu
NEG mem16/32	F7h		mm-011-xx	vector	
NOP (XCHG AX, AX)	90h			short	limm
NOT mreg8	F6h		11-010-xxx	short	alux
NOT mem8	F6h		mm-010-xx	vector	
NOT mreg16/32	F7h		11-010-xxx	short	alu
NOT mem16/32	F7h		mm-010-xx	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		xx-xxx-xxx	short	alux
OR EAX, imm16/32	0Dh		xx-xxx-xxx	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
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® Opcodes
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	8Fh		11-000-xxx	short	load alu
POP mem	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

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	
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	

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	
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	

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	
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		xx-xxx-xxx	vector	
SBB EAX, imm16/32	1Dh		xx-xxx-xxx	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	

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	
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

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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
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	

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
STI	FBh			vector	
STOSB mem8, AL	AAh			long	store alux
STOSW mem16, AX	ABh			long	store alux
STOSD mem32, EAX	ABh			long	store alux
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
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		xx-xxx-xxx	short	alux
SUB EAX, imm16/32	2Dh		xx-xxx-xxx	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
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-xx	long	load, alux
TEST mreg8, imm16/32	F7h		11-000-xxx	long	alu

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
TEST mem8, imm16/32	F7h		mm-000-xx	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	
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		xx-xxx-xxx	short	alux
XOR EAX, imm16/32	35h		xx-xxx-xxx	short	alu

Table 6. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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
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® Opcodes
EMMS	0Fh	77h		vector	
MOVD mmreg, mreg32	0Fh	6Eh	11-xxx-xxx	short	store, mload
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 mmreg1, mmreg2	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

Table 7. MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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
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	11-xxx-xxx	short	mload, meu

Table 7. MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	modR/M Byte	Decode Type	RISC86® Opcodes
PSLLW mmreg, imm8	0Fh	71h	11-110-xxx	short	meu
PSLLD mmreg1, mmreg2	0Fh	F2h	11-xxx-xxx	short	meu
PSLLD mmreg, mem64	0Fh	F2h	11-xxx-xxx	short	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	11-xxx-xxx	short	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	11-xxx-xxx	short	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	11-xxx-xxx	short	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	11-xxx-xxx	short	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	11-xxx-xxx	short	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	11-xxx-xxx	short	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
PSUBUSW mmreg1, mmreg2	0Fh	D9h	11-xxx-xxx	short	meu

Table 7. MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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

Table 8. Floating-Point Instructions

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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

Table 8. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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		11-011-001	short	float
FCOS ST(0)	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
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

Table 8. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	
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
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

Table 8. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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
FPTAN	D9h	F2h		vector	
FPATAN	D9h	F3h		short	float
FPREM	D9h	F8h		short	float
FPREM1	D9h	F5h		short	float
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
FST mem32real	D9h		mm-010-xxx	short	fstore
FST mem64real	DDh		mm-010-xxx	short	fstore
FST ST(i)	DDh		11-010xxx	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

Table 8. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	modR/M Byte	Decode Type	RISC86® Opcodes
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	

5

x86 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 MMX enhanced processor, 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 processor or are not required and have no effect (due to fewer coding restrictions with the AMD-K6 processor).

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/op/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/op/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.

Code Alignment—Aligning at 0-mod-16 improves performance (ideally at 0-mod-32). However, there is a trade-off between execution speed and code size.

General AMD-K6™ Processor x86 Coding Optimizations

This section describes general code optimization techniques specific to the AMD-K6 MMX enhanced processor.

Use short-decodeable instructions—To increase decode bandwidth and minimize the number of RISC86 operations per x86 instruction, use short-decodeable x86 instructions. See "Instruction Dispatch and Execution Timing" on page 23 for the list of short-decodeable instructions.

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

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 than short-decodeable or long-decodeable instructions.

0Fh prefix usage—0Fh does not count as a prefix.

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.

Align branch targets—Keep branch targets away from the end of a cache line. 16-byte alignment is preferred for branch targets, while 32-byte alignment is ideal.

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 pollution with rarely used items.

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 at a time until the instruction is completely decoded. Table 9 shows when prefixes are accumulated and decoding is serialized.

Table 9. 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
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

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 MMX enhanced 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:

```
Avoid OR    AH,055h
      AND   EAX,1555555h
```

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 that can replace straight forward code with data dependent branches. 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
```

The [ESI] addressing mode—When using [ESI] as an indirect memory addressing mode, explicitly code [ESI] to be [ESI+0]. Doing so improves decode bandwidth. For an example, see "Floating-Point Code Sample" on page 64.

AMD-K6™ Processor Integer x86 Coding Optimizations

This section describes integer code optimization techniques specific to the AMD-K6 MMX enhanced processor.

Neutral code filler—Use the XCHG EAX, EAX or NOP instruction when aligning instructions. XCHG EAX, EAX consumes decode slots 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.

Access 16-bit memory data using the MOVSX and MOVZX instructions—The AMD-K6 processor has direct hardware support for extending word size operands to doubleword length.

Use load-execute integer instructions—Most load-execute integer instructions are short-decodeable 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.

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 MMX enhanced 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], 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 is a commonly used optimization on RISC CPUs. 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™ Processor Multimedia Coding Optimizations

This section describes multimedia code optimization techniques specific to the AMD-K6 MMX enhanced processor.

Pair MMX instructions with short-decodeable instructions— MMX instructions are short-decodeable and can be simultaneously decoded with any other short-decodeable instruction. This technique requires that MMX instructions be arranged as the first of a pair of short-decodeable instructions.

Avoid using MMX registers to move double-precision floating-point data—Although using an MMX register to move floating-point data appears fast, using MMX registers requires the use of the EMMS instruction when switching from MMX to floating-point instructions.

Avoid switching between MMX and FPU instructions—Because the MMX registers are mapped on to the floating-point stack, the EMMS instruction must be executed after using MMX code and prior to the use of the floating-point unit. Group or partition MMX code away from FPU code so that the use of the EMMS instruction is minimized. Also, the actual penalty from the use of the EMMS instruction occurs not at the time of execution but when the first floating-point instruction is encountered.

AMD-K6™ Processor Floating-Point Coding Optimizations

This section describes floating-point code optimization techniques specific to the AMD-K6 MMX enhanced processor.

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

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

Avoid FXCH usage—Pairing FXCH with other floating-point instructions does not increase performance.

Avoid switching between MMX and FPU instructions—Because the MMX registers are mapped on to the floating-point stack, the EMMS instruction must be executed after using MMX code and prior to the use of the floating-point unit. Group or partition MMX code away from FPU code so that the use of the EMMS instruction is minimized. Also, the actual penalty from the use of the EMMS instruction occurs not at the time of execution but when the first floating-point instruction is encountered.

Avoid using MMX registers to move double-precision floating-point data—Although using an MMX register to move floating-point data appears fast, using MMX registers requires the use of the EMMS instruction when switching from MMX to floating-point instructions.

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

Avoid splitting floating-point instructions with integer instructions—No penalty is incurred when using arithmetic or comparison floating-point instructions that use integer operands, such as FIADD or FICOM. 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. When 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 AMD-K6 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 example below shows how to translate the C-style code when moving double-precision floating-point data:

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

Avoid: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 AMD-K6 processor has a low-latency, non-pipelined floating-point execution unit. Therefore, no scheduling between floating-point instructions is necessary.

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 FP reg, reg instruction. For the AMD-K6 processor, load-execute arithmetic and compare instructions

are identical in throughput to FP 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 important rules to optimize this matrix multiply routine. The first rule is to force [ESI] to be [ESI+0]. The second rule is the insertion of NOPs to avoid cache line straddles. The third rule used is avoiding 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 AMD-K6 MMX enhanced processor-specific optimization techniques affect other processors, including the AMD-K5 processor.

Table 10. 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 decodeable 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 only if the upper half of the product is produced)

Table 10. 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-decodeable 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.
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-decodeable and consume decode bandwidth but no execution resources.

Table 10. 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
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 11. 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	
Index Versus Base Register	1-cycle penalty to calculate the effective address when an index register is used.	None	

Table 11. 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
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-decodeable, 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 decodeable 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 decodeable only.
Misaligned Data Penalty	3-clock delay	Partial	1-clock delay.

Table 12. 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	MULL - 1 clock MULH - 2 clocks MADD - 2 clocks
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	Pairing requires short-decodeable integer instruction as the second instruction.
U Pipe: MMX/Integer Pairing	V pipe integer instruction must be pairable.	Similar	Pairing requires short-decodeable integer instruction as the second instruction.
Pairing Two MMX Instructions	Cannot pair two MMX multiplies, two MMX shifts, or MMX instructions in V pipe with U pipe dependency.	None	
66h or 67h Prefix Penalty	Three clocks.	None	

Table 13. AMD-K6™ Processor and Pentium® Pro Processor-Specific Optimizations

Pentium Pro Guideline/Event	Pentium Pro 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 P6 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.	Different	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 13. AMD-K6™ Processor and Pentium® Pro Processor-Specific Optimizations (continued)

Pentium Pro Guideline/Event	Pentium Pro Effect	Usage/Effect on AMD-K6 Processor	AMD-K6 Processor Detail
Branches	Exploit the P6 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 P6 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	Different	1-clock maximum delay.
2-Byte Data Alignment	4-byte boundary	Same	Note, the misalignment penalty is only a 1-clock delay.
4-Byte Data Alignment	4-byte boundary	Same	Note, the misalignment penalty is only a 1-clock delay.
8-Byte Data Alignment	8-byte boundary	Same	Note, the misalignment penalty is only a 1-clock delay.
Instruction Lengths Greater Than 7 Bytes	Issued one at a time	Different	Long-decodeable and vector-decodeable.
Prefix Penalty	1-clock delay	Possible	Delays can sometimes occur due to prefix accumulation.
0Fh Prefix Penalty	None	None	
MOVZX Acceleration	Yes	Yes	Short-decodeable, 1 cycle.
Static Prediction Penalty	6 clocks	Different	3 clocks.

Table 14. AMD-K6™ Processor and Pentium® Pro with Optimizations for MMX™ Instructions

Pentium Pro Guideline/Event	Pentium Pro 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	MULL - 1 clock MULH - 2 clocks MADD - 2 clocks
Pairing Two MMX Instructions	Cannot pair two MMX multiplies, two MMX shifts, or MMX instructions in V-pipe with U-pipe dependency.	None	
Predicted Branches not in the BTB	~5-cycle latency	Different	1-cycle latency for BTB miss.