AMD Athlon™ Processor

x86 Code Optimization Guide

Publication # **22007** Rev: **D**
Issue Date: **August 1999**

Contents

Revision History xv

1 Introduction 1

AMD Athlon™ Processor Family 2

AMD Athlon Processor 2

2 Top Optimizations 5

Optimization Star 6

Group I Optimizations — Essential Optimizations 6

Memory Size and Alignment Issues 6

Use the 3DNow!™ PREFETCH and PREFETCHW Instructions 6

Select DirectPath Over VectorPath Instructions 7

Group II Optimizations—Secondary Optimizations 7

Load-Execute Instruction Usage 7

Take Advantage of Write Combining 8

Use 3DNow! Instructions 8

Avoid Branches Dependent on Random Data 8

Avoid Placing Code and Data in the Same 64-Byte Cache Line 9

3 C Source Level Optimizations 11

Ensure Floating-Point Variables and Expressions are of Type Float 11

Use 32-Bit Data Types for Integer Code 11

Use Unsigned Integer Types over Signed Integer Types 12

Completely Unroll Small Loops 12

Avoid Unnecessary Store-to-Load Dependencies 13

Switch Statement Usage 14

Optimize Switch Statements 14

Use Prototypes for All Functions 15

Use Const Type Qualifier 15
Generic Loop Hoisting 15
Generalization for Multiple Constant Control Code. 16
Declare Local Functions as Static 18
Dynamic Memory Allocation Consideration 18
Introduce Explicit Parallelism into Code 19
Explicitly Extract Common Subexpressions 20
C Language Structure Component Considerations 21
Sort Local Variables According to Base Type Size 22
Avoid Unnecessary Integer Division. 23
Copy Frequently De-referenced Pointer Arguments to Local Variables 23

4 Instruction Decoding Optimizations 25

Overview 25
Select DirectPath Over VectorPath Instructions. 26
Load-Execute Instruction Usage 26
Use Load-Execute Integer Instructions 26
Use Load-Execute Floating-Point Instructions with Floating-Point Operands 26
Avoid Load-Execute Floating-Point Instructions with Integer Operands 27
Align Branch Targets 28
Use Short Instruction Lengths. 28
Avoid Partial Register Reads and Writes. 29
Replace Certain SHLD Instructions with Alternative Code. 29
Use 8-Bit Sign-Extended Immediates 30
Use 8-Bit Sign-Extended Displacements. 30

Code Padding Using Neutral Code Fillers 31
Recommendations for the AMD Athlon™ Processor 31
Recommendations for AMD-K6® Family and AMD Athlon Processor Blended Code 32

5 Cache and Memory Optimizations 37

Memory Size and Alignment Issues 37
Avoid Memory Size Mismatches 37
Align Data Where Possible 38
Use the 3DNow! PREFETCH and PREFETCHW Instructions 38
Take Advantage of Write Combining 41
Avoid Placing Code and Data in the Same 64-Byte Cache Line 42
Store-to-Load Forwarding Restrictions 42
Store-to-Load Forwarding Pitfalls—True Dependencies 43
Summary of Store-to-Load Forwarding Pitfalls to Avoid 45
Stack Alignment Considerations 46
Align TBYTE Variables on Quadword Aligned Addresses 47
C Language Structure Component Considerations 47
Sort Variables According to Base Type Size 47

6 Branch Optimizations 49

Avoid Branches Dependent on Random Data 49
AMD Athlon Processor Specific Code 50
Blended AMD-K6 and AMD Athlon Processor Code 50
Always Pair CALL and RETURN 51
Replace Branches with Computation in 3DNow! Code 52
Muxing Constructs 52
Sample Code Translated Into 3DNow! Code 53
Avoid the Loop Instruction 56
Avoid Far Control Transfer Instructions 56
Avoid Recursive Functions 57

7 Scheduling Optimizations 59

Schedule Instructions According to Their Latency 59
Unrolling Loops 59
Complete Loop Unrolling 59
Partial Loop Unrolling 60
Use Function Inlining 62
Overview 62
Always Inline Functions If Called From One Site 63
Always Inline Functions with Fewer Than 25 Machine Instructions 64
Avoid Address Generation Interlocks 64
Use MOVZX and MOVSX 65
Minimize Pointer Arithmetic in Loops 65
Push Memory Data Carefully 67

8 Integer Optimizations 69

Replace Divides with Multiplies 69
Multiplication by Reciprocal (Division) Utility 69
Unsigned Division by Multiplication of Constant 70
Signed Division by Multiplication of Constant 71
Use Alternative Code When Multiplying by a Constant 73
Use MMX Instructions for Integer-Only Work 75
Repeated String Instruction Usage 76
Latency of Repeated String Instructions 76
Guidelines for Repeated String Instructions 76
Use MOVQs for Moving a Quadword Aligned Block of Data 77
Use XOR Instruction to Clear Integer Registers 78
Efficient 64-Bit Integer Arithmetic 78

Derivation of Multiplier Used For Integer Division by Constants. . 84
Unsigned Derivation for Algorithm, Multiplier, and Shift Factor 84
Signed Derivation for Algorithm, Multiplier, and Shift Factor 86

9 Floating-Point Optimizations 89

Ensure All FPU Data is Aligned 89
Use Multiplies Rather Than Divides. 89
Use FFREEP Macro to Pop One Register from the FPU Stack 90
Floating-Point Compare Instructions 90
Use the FXCH Instruction Rather Than FST/FLD Pairs 91
Avoid Using Extended-Precision Data 91
Minimize Floating-Point-to-Integer Conversions 91
Floating-Point Subexpression Elimination. 94

10 3DNow!™ and MMX™ Optimizations 95

Use 3DNow! Instructions 95
Use FEMMS Instruction 95
Use 3DNow! Instructions for Fast Division 96
Optimized 15-Bit Precision Divide 96
Optimized Full 24-Bit Precision Divide 96
Pipelined Pair of 24-Bit Precision Divides. 96
Newton-Raphson Reciprocal 97
Use 3DNow! Instructions for Fast Square Root and Reciprocal Square Root 98
Optimized 15-Bit Precision Square Root 98
Optimized 24-Bit Precision Square Root 98
Newton-Raphson Reciprocal Square Root. 99
Use MMX PMADDWD Instruction to Perform Two 32-Bit Multiplies in Parallel 99
3DNow! and MMX Intra-Operand Swapping 100

Fast Conversion of Signed Words to Floating-Point 100
Use MMX PXOR to Change the Sign Bit in 3DNow! Code 101
Use MMX PCMP Instead of 3DNow! PFCMP. 102
Use MMX PXOR to Clear an MMX Register 102
Use MMX PCMPEQD to Set an MMX Register 103
Use MMX PAND to Find Absolute Value in 3DNow! Code 103
Use MMX PXOR to Negate 3DNow! Data 103
Use 3DNow! PAVGUSB for MPEG-2 Motion Compensation 103
Stream of Packed Unsigned Bytes 105

11 General x86 Optimization Guidelines 107

Short Forms 107
Dependencies 108
Register Operands 108
Stack Allocation 108

Appendix A AMD Athlon™ Processor Microarchitecture 109

Introduction 109
AMD Athlon Processor Microarchitecture 110
Superscalar Processor 110
Instruction Cache 111
Predecode 112
Branch Prediction. 112
Early Decoding 113
Instruction Control Unit 113
Data Cache 114
Integer Scheduler. 115
Integer Execution Unit 115
Floating-Point Scheduler. 116
Floating-Point Execution Unit 116
Load-Store Unit (LSU). 117
L2 Cache Controller 118

Write Combining 118
AMD Athlon System Bus 119

Appendix B Pipeline and Execution Unit Resources Overview 121

Fetch and Decode Pipeline Stages 121
Integer Pipeline Stages 124
Floating-Point Pipeline Stages 126
Execution Unit Resources 127
Terminology 127
Integer Pipeline Operations 129
Floating-Point Pipeline Operations 130
Load/Store Pipeline Operations 131
Code Sample Analysis 132

Appendix C Implementation of Write Combining 135

Introduction 135
Write-Combining Definitions and Abbreviations 136
What is Write Combining? 136
Programming Details 136
Write-Combining Operations 137
Sending Write-Buffer Data to the System 139

Appendix D Instruction Dispatch and Execution Timing 141

Appendix E DirectPath versus VectorPath Instructions 175

Select DirectPath Over VectorPath Instructions. 175
DirectPath Instructions 175
VectorPath Instructions 187

Appendix F Performance Monitoring Counters 193

Overview 193

Performance Counter Usage 193

PerfEvtSel[3:0] MSRs (MSR Addresses C001_0000h–C001_0003h) 194

PerfCtr[3:0] MSRs (MSR Addresses C001_0004h–C001_0007h) 199

Starting and Stopping the Performance-Monitoring Counters 200

Event and Time-Stamp Monitoring Software 200

Monitoring Counter Overflow. 201

List of Figures

Figure 1. AMD Athlon™ Processor Block Diagram 111
Figure 2. Integer Execution Pipeline 115
Figure 3. Floating-Point Unit Block Diagram 117
Figure 4. Load/Store Unit 118
Figure 5. Fetch/Scan/Align/Decode Pipeline Hardware 122
Figure 6. Fetch/Scan/Align/Decode Pipeline Stages 122
Figure 7. Integer Execution Pipeline 124
Figure 8. Integer Pipeline Stages 124
Figure 9. Floating-Point Unit Block Diagram 126
Figure 10. Floating-Point Pipeline Stages 126
Figure 11. PerfEvtSel[3:0] Registers 194

List of Tables

Table 1. Latency of Repeated String Instructions. 76
Table 2. Integer Pipeline Operation Types 129
Table 3. Integer Decode Types 129
Table 4. Floating-Point Pipeline Operation Types 130
Table 5. Floating-Point Decode Types 130
Table 6. Load/Store Unit Stages 131
Table 7. Sample 1 – Integer Register Operations 133
Table 8. Sample 2 – Integer Register and Memory Load Operations. 134
Table 9. Write Combining Completion Events 138
Table 10. AMD Athlon™ System Bus Commands Generation Rules 139
Table 11. Integer Instructions 142
Table 12. MMX™ Instructions. 162
Table 13. MMX Extensions 165
Table 14. Floating-Point Instructions 166
Table 15. 3DNow!™ Instructions. 171
Table 16. 3DNow! Extensions 173
Table 17. DirectPath Integer Instructions 176
Table 18. DirectPath MMX Instructions. 183
Table 19. DirectPath MMX Extensions. 184
Table 20. DirectPath Floating-Point Instructions 185
Table 21. VectorPath Integer Instructions. 187
Table 22. VectorPath MMX Instructions 190
Table 23. VectorPath MMX Extensions 190
Table 24. VectorPath Floating-Point Instructions. 191
Table 26. Performance Monitoring Counters. 196

Revision History

Date	Rev	Description
August 1999	D	Initial public release

1

Introduction

The AMD Athlon™ processor is the newest microprocessor in the AMD K86™ family of microprocessors. The advances in the AMD Athlon processor take superscalar operation and out-of-order execution to a new level. The AMD Athlon processor has been designed to efficiently execute code written for previous-generation x86 processors. However, to enable the fastest code execution with the AMD Athlon processor, programmers should write software that includes specific code optimization techniques.

This document contains information to assist programmers in creating optimized code for the AMD Athlon processor. In addition, this document is targeted at compiler and assembler designers and assembly language programmers writing execution-sensitive code sequences.

This document assumes that the reader possesses in-depth knowledge of the x86 instruction set, the x86 architecture (registers, programming modes, etc.), and the IBM PC-AT platform.

This guide has been written specifically for the AMD Athlon processor, but it includes considerations for previous-generation processors and how those optimizations are applicable to the AMD Athlon processor.

AMD Athlon™ Processor Family

The AMD Athlon processor family uses state-of-the-art decoupled decode/execution design techniques to deliver next-generation performance with x86 binary software compatibility. This next-generation processor family advances x86 code execution by using flexible instruction predecoding, wide and balanced decoders, aggressive out-of-order execution, parallel integer execution pipelines, parallel floating-point execution pipelines, deep pipelined execution for higher delivered operating frequency, dedicated backside cache memory, and a new high-performance double-rate 64-bit local bus. As an x86 binary-compatible processor, the AMD Athlon processor implements the industry-standard x86 instruction set by decoding and executing the x86 instructions using a proprietary microarchitecture. This microarchitecture allows the delivery of maximum performance when running x86-based PC software.

AMD Athlon™ Processor

The AMD Athlon processor brings superscalar performance and high operating frequency to PC systems running industry-standard x86 software. A brief summary of the next-generation design features implemented in the AMD Athlon processor is as follows:

- High-speed double-rate local bus interface
- Large, split 128-Kbyte level-one (L1) cache
- Dedicated backside level-two (L2) cache
- Instruction predecode and branch detection during cache line fills
- Decoupled decode/execution core
- Three-way x86 instruction decoding
- Dynamic scheduling and speculative execution
- Three-way integer execution
- Three-way address generation
- Three-way floating-point execution

- 3DNow!™ technology and MMX™ single-instruction multiple-data (SIMD) instruction extensions
- Super data forwarding
- Deep out-of-order integer and floating-point execution
- Register renaming
- Dynamic branch prediction

The AMD Athlon processor communicates through a next-generation high-speed local bus that is beyond the current Socket 7 or Super7™ bus standard. The local bus can transfer data at twice the rate of the bus operating frequency by using both the rising and falling edges of the clock (see "AMD Athlon™ System Bus" on page 119 for more information).

To reduce on-chip cache miss penalties and to avoid subsequent data load or instruction fetch stalls, the AMD Athlon processor has a dedicated high-speed backside L2 cache. The large 128-Kbyte L1 on-chip cache and the backside L2 cache allow the AMD Athlon execution core to achieve and sustain maximum performance.

As a decoupled decode/execution processor, the AMD Athlon processor makes use of a proprietary microarchitecture, which defines the heart of the AMD Athlon processor. With the inclusion of all these features, the AMD Athlon processor is capable of decoding, issuing, executing, and retiring multiple x86 instructions per cycle, resulting in superior scaleable performance.

The AMD Athlon processor includes both the industry-standard MMX SIMD integer instructions and the 3DNow! SIMD floating-point instructions that were first introduced in the AMD-K6®-2 processor. The design of 3DNow! technology was based on suggestions from leading graphics and independent software vendors (ISVs). Using SIMD format, the AMD Athlon processor can generate up to four 32-bit, single-precision floating-point results per clock cycle.

The 3DNow! execution units allow for high-performance floating-point vector operations, which can replace x87 instructions and enhance the performance of 3D graphics and other floating-point-intensive applications. Because the 3DNow! architecture uses the same registers as the MMX

instructions, switching between MMX and 3DNow! has no penalty.

The AMD Athlon processor designers took another innovative step by carefully integrating the traditional x87 floating-point, MMX, and 3DNow! execution units into one operational engine. With the introduction of the AMD Athlon processor, the switching overhead between x87, MMX, and 3DNow! technology is virtually eliminated. The AMD Athlon processor combined with 3DNow! technology brings a better multimedia experience to mainstream PC users while maintaining backwards compatibility with all existing x86 software.

Although the AMD Athlon processor can extract code parallelism on-the-fly from off-the-shelf, commercially available x86 software, specific code optimization for the AMD Athlon processor can result in even higher delivered performance. This document describes the proprietary microarchitecture in the AMD Athlon processor and makes recommendations for optimizing execution of x86 software on the processor.

The coding techniques for achieving peak performance on the AMD Athlon processor include, but are not limited to, those for the AMD-K6, AMD-K6-2, Pentium®, Pentium Pro, and Pentium II processors. However, many of these optimizations are not necessary for the AMD Athlon processor to achieve maximum performance. Due to the more flexible pipeline control and aggressive out-of-order execution, the AMD Athlon processor is not as sensitive to instruction selection and code scheduling. This flexibility is one of the distinct advantages of the AMD Athlon processor.

The AMD Athlon processor uses the latest in processor microarchitecture design techniques to provide the highest x86 performance for today's PC. In short, the AMD Athlon processor offers true next-generation performance with x86 binary software compatibility.

2

Top Optimizations

This chapter contains concise descriptions of the best optimizations for improving the performance of the AMD Athlon™ processor. Subsequent chapters contain more detailed descriptions of these and other optimizations. The optimizations in this chapter are divided into two groups and listed in order of importance.

Group I – Essential Optimizations

Group I contains essential optimizations. Users should follow these critical guidelines closely. The optimizations in Group I are as follows:

- **Memory Size and Alignment Issues**—Avoid memory size mismatches—Align data where possible
- **Use the 3DNow! PREFETCH and PREFETCHW Instructions**
- **Select DirectPath Over VectorPath Instructions**

Group II – Secondary Optimizations

Group II contains secondary optimizations that can significantly improve the performance of the AMD Athlon processor. The optimizations in Group II are as follows:

- **Load-Execute Instruction Usage**—Use Load-Execute instructions—Avoid load-execute floating-point instructions with integer operands
- **Take Advantage of Write Combining**
- **Use 3DNow!™ Instructions**
- **Avoid Branches Dependent on Random Data**

- **Avoid Placing Code and Data in the Same 64-Byte Cache Line**

Optimization Star

The top optimizations described in this chapter are flagged with a star. In addition, the star appears beside the more detailed descriptions found in subsequent chapters.

Group I Optimizations – Essential Optimizations

Memory Size and Alignment Issues

See "Memory Size and Alignment Issues" on page 37 for more details.

Avoid Memory Size Mismatches

Avoid memory size mismatches when instructions operate on the same data. For instructions that store and reload the same data, keep operands aligned and keep the loads/stores of each operand the same size.

Align Data Where Possible

Avoid misaligned data references. A misaligned store or load operation suffers a minimum one-cycle penalty in the AMD Athlon processor load/store pipeline.

Use the 3DNow!™ PREFETCH and PREFETCHW Instructions

For code that can take advantage of prefetching, use the 3DNow! PREFETCH and PREFETCHW instructions to increase the effective bandwidth to the AMD Athlon processor, which significantly improves performance. Use the following formula to determine prefetch distance:

Prefetch Length = 200 ($^{DS}/_{C}$)

- Round up to the nearest cache line.
- DS is the data stride per loop iteration.
- C is the number of cycles per loop iteration when hitting in the L1 cache.

See "Use the 3DNow!™ PREFETCH and PREFETCHW Instructions" on page 38 for more details.

Select DirectPath Over VectorPath Instructions

Use DirectPath instructions rather than VectorPath instructions. DirectPath instructions are optimized for decode and execute efficiently by minimizing the number of operations per x86 instruction. Using VectorPath instructions may block DirectPath instructions from decoding simultaneously.

See Appendix E, "DirectPath versus VectorPath Instructions" on page 175 for a list of DirectPath and VectorPath instructions.

Group II Optimizations—Secondary Optimizations

Load-Execute Instruction Usage

See "Load-Execute Instruction Usage" on page 26 for more details.

Use Load-Execute Instructions

Wherever possible, use load-execute instructions to increase code density with the one exception described below. The split-instruction form of load-execute instructions can be used to avoid scheduler stalls for longer executing instructions and to explicitly schedule the load and execute operations.

Avoid Load-Execute Floating-Point Instructions with Integer Operands

Do not use load-execute floating-point instructions with **integer operands**. The floating-point load-execute instructions with integer operands are VectorPath and generate two OPs in a cycle, while the discrete equivalent enables a third DirectPath instruction to be decoded in the same cycle.

Take Advantage of Write Combining

This guideline applies only to operating system, device driver, and BIOS writers. In order to improve system performance, the AMD Athlon processor aggressively combines multiple memory-write cycles of any data size that address locations within a 64-byte cache line aligned write buffer.

See Appendix C, “Implementation of Write Combining” on page 135 for more details.

Use 3DNow!™ Instructions

Unless accuracy requirements dictate otherwise, perform floating-point computations using the 3DNow! instructions instead of x87 instructions. The SIMD nature of 3DNow! instructions achieves twice the number of FLOPs that are achieved through x87 instructions. 3DNow! instructions also provide for a flat register file instead of the stack-based approach of x87 instructions.

See Table 15 on page 171 for a list of 3DNow! instructions. For information about instruction usage, see the *3DNow!™ Technology Manual*, order# 21928.

Avoid Branches Dependent on Random Data

Avoid data-dependent branches around a single instruction. Data-dependent branches acting upon basically random data can cause the branch prediction logic to mispredict the branch about 50% of the time. Design branch-free alternative code sequences, which results in shorter average execution time.

See “Avoid Branches Dependent on Random Data” on page 49 for more details.

Avoid Placing Code and Data in the Same 64-Byte Cache Line

Consider that the AMD Athlon processor cache line is twice the size of previous processors. Code and data should not be shared in the same 64-byte cache line, especially if the data ever becomes modified. In order to maintain cache coherency, the AMD Athlon processor may thrash its caches, resulting in lower performance.

In general the following should be avoided:

- Self-modifying code
- Storing data in code segments

See “Avoid Placing Code and Data in the Same 64-Byte Cache Line” on page 42 for more details.

3

C Source Level Optimizations

This chapter details C programming practices for optimizing code for the AMD Athlon™ processor. Guidelines are listed in order of importance.

Ensure Floating-Point Variables and Expressions are of Type Float

For compilers that generate 3DNow!™ instructions, make sure that all floating-point variables and expressions are of type float. Pay special attention to floating-point constants. These require a suffix of "F" or "f" (for example, 3.14f) in order to be of type float, otherwise they default to type double. To avoid automatic promotion of float arguments to double, always use function prototypes for all functions that accept float arguments.

Use 32-Bit Data Types for Integer Code

Use 32-bit data types for integer code. Compiler implementations vary, but typically the following data types are included—*int, signed, signed int, unsigned, unsigned int, long, signed long, long int, signed long int, unsigned long,* and *unsigned long int*.

Use Unsigned Integer Types over Signed Integer Types

If possible, use unsigned integer types over signed integer types. The unsigned types convey to the compiler that data cannot be negative, which allows some optimizations not possible with signed and potentially negative data.

In most programs, certain variables have to be of signed types due to the nature of the data stored in them (for example, temperatures). In some cases, aggressive use of unsigned types can create many mixed expressions containing both signed and unsigned terms. It can be difficult to determine the exact semantics of such expressions.

Completely Unroll Small Loops

Take advantage of the AMD Athlon processor's large, 64-Kbyte instruction cache and completely unroll small loops. Unrolling loops can be beneficial to performance, especially if the loop body is small and the loop overhead is, therefore, significant. Many compilers are not aggressive at unrolling loops. For loops that have a small fixed loop count and a small loop body, completely unrolling the loops at the source level is recommended.

Example 1 (Avoid):

```
// 3D-transform: multiply vector V by 4x4 transform matrix M
for (i=0; i<4; i++) {
   r[i] = 0;
   for (j=0; j<4; j++) {
      r[i] += M[j][i]*V[j];
   }
}
```

Example 2 (Preferred):

```
// 3D-transform: multiply vector V by 4x4 transform matrix M
r[0] =  M[0][0]*V[0] + M[1][0]*V[1] + M[2][0]*V[2] +
        M[3][0]*V[3];
r[1] =  M[0][1]*V[0] + M[1][1]*V[1] + M[2][1]*V[2] +
        M[3][1]*V[3];
r[2] =  M[0][2]*V[0] + M[1][2]*V[1] + M[2][2]*V[2] +
        M[3][2]*V[3];
r[3] =  M[0][3]*V[0] + M[1][3]*V[1] + M[2][3]*V[2] +
        M[3][3]*v[3];
```

Avoid Unnecessary Store-to-Load Dependencies

A store-to-load dependency exists when data is stored to memory, only to be read back shortly thereafter. See "Store-to-Load Forwarding Restrictions" on page 42 for more details. The AMD Athlon processor contains hardware to accelerate such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory. However, it is still faster to avoid such dependencies altogether and keep the data in an internal register.

Avoiding store-to-load dependencies is especially important if they are part of a long dependency chains, as might occur in a recurrence computation. If the dependency occurs while operating on arrays, many compilers are unable to optimize the code in a way that avoids the store-to-load dependency. In some instances the language definition may prohibit the compiler from using code transformations that would remove the store-to-load dependency. It is therefore recommended that the programmer remove the dependency manually, e.g., by introducing a temporary variable that can be kept in a register. This can result in a significant performance increase. The following is an example of this.

Example 1 (Avoid):

```
double x[VECLEN], y[VECLEN], z[VECLEN];
unsigned int k;

for (k = 1; k < VECLEN; k++) {
   x[k] = x[k-1] + y[k];
}

for (k = 1; k < VECLEN; k++) {
   x[k] = z[k] * (y[k] - x[k-1]);
}
```

Example 2 (Preferred):

```
double x[VECLEN], y[VECLEN], z[VECLEN];
unsigned int k;
double t;

t = x[0];
for (k = 1; k < VECLEN; k++) {
   t = t + y[k];
   x[k] = t;
}
```

```
t = x[0];
for (k = 1; k < VECLEN; k++) {
   t = z[k] * (y[k] - t);
   x[k] = t;
}
```

Switch Statement Usage

Optimize Switch Statements

Switch statements are translated using a variety of algorithms. The most common of these are jump tables and comparison chains/trees. It is recommended to sort the cases of a switch statement according to the probability of occurrences, with the most probable first. This will improve performance when the switch is translated as a comparison chain. It is further recommended to make the case labels small, contiguous integers, as this will allow the switch to be translated as a jump table.

Example 1 (Avoid):

```
int days_in_month, short_months, normal_months, long_months;

switch (days_in_month) {
  case 28:
  case 29: short_months++; break;
  case 30: normal_months++; break;
  case 31: long_months++; break;
  default: printf ("month has fewer than 28 or more than 31
days\n");
}
```

Example 2 (Preferred):

```
int days_in_month, short_months, normal_months, long_months;

switch (days_in_month) {
  case 31: long_months++; break;
  case 30: normal_months++; break;
  case 28:
  case 29: short_months++; break;
  default: printf ("month has fewer than 28 or more than 31
days\n");
}
```

Use Prototypes for All Functions

In general, use prototypes for all functions. Prototypes can convey additional information to the compiler that might enable more aggressive optimizations.

Use Const Type Qualifier

Use the “const” type qualifier as much as possible. This optimization makes code more robust and may enable higher performance code to be generated due to the additional information available to the compiler. For example, the C standard allows compilers to not allocate storage for objects that are declared “const”, if their address is never taken.

Generic Loop Hoisting

To improve the performance of inner loops, it is beneficial to reduce redundant constant calculations (i.e., loop invariant calculations). However, this idea can be extended to invariant control structures.

The first case is that of a constant “if()” statement in a “for()” loop.

Example 1:

```
for( i ... ) {
   if( CONSTANT0 ) {
         DoWork0( i );     // does not affect CONSTANT0
   } else {
         DoWork1( i );     // does not affect CONSTANT0
   }
}
```

The above loop should be transformed into:

```
if( CONSTANT0 ) {
   for( i ... ) {
         DoWork0( i );
   }
} else {
   for( i ... ) {
         DoWork1( i );
   }
}
```

This will make your inner loops tighter by avoiding repetitious evaluation of a known "if()" control structure. Although the branch would be easily predicted, the extra instructions and decode limitations imposed by branching are saved, which are usually well worth it.

Generalization for Multiple Constant Control Code

To generalize this further for multiple constant control code some more work may have to be done to create the proper outer loop. Enumeration of the constant cases will reduce this to a simple switch statement.

Example 2:

```
for(i ... ) {
   if( CONSTANT0 ) {
         DoWork0( i );     //does not affect CONSTANT0
                           // or CONSTANT1
   } else {
         DoWork1( i );     //does not affect CONSTANT0
                           // or CONSTANT1
   }
   if( CONSTANT1 ) {
         DoWork2( i );     //does not affect CONSTANT0
                           // or CONSTANT1
   } else {
         DoWork3( i );     //does not affect CONSTANT0
                           // or CONSTANT1
   }
}
```

The above loop should be transformed into:

```
#define combine( c1, c2 ) (((c1) << 1) + (c2))
switch( combine( CONSTANT0!=0, CONSTANT1!=0 ) ) {
          case combine( 0, 0 ):
                for( i ... ) {
                      DoWork0( i );
                      DoWork2( i );
                }
                break;
          case combine( 1, 0 ):
                for( i ... ) {
                      DoWork1( i );
                      DoWork2( i );
                }
                break;
          case combine( 0, 1 ):
                for( i ... ) {
                      DoWork0( i );
                      DoWork3( i );
                }
                break;
          case combine( 1, 1 ):
                for( i ... ) {
                      DoWork1( i );
                      DoWork3( i );
                }
                break;
          default:
                break;
}
```

The trick here is that there is some up-front work involved in generating all the combinations for the switch constant and the total amount of code has doubled. However, it is also clear that the inner loops are "if()-free". In ideal cases where the “DoWork*()” functions are inlined, the successive functions will have greater overlap leading to greater parallelism than would be possible in the presence of intervening “if()” statements.

The same idea can be applied to constant “switch()” statements, or combinations of “switch()” statements and “if()” statements inside of “for()” loops. The method for combining the input constants gets more complicated but will be worth it for the performance benefit.

However, the number of inner loops can also substantially increase. If the number of inner loops is prohibitively high, then

only the most common cases need to be dealt with directly, and the remaining cases can fall back to the old code in a "default:" clause for the “switch()” statement.

This typically comes up when the programmer is considering runtime generated code. While runtime generated code can lead to similar levels of performance improvement, it is much harder to maintain, and the developer must do their own optimizations for their code generation without the help of an available compiler.

Declare Local Functions as Static

Functions that are not used outside the file in which they are defined should always be declared static, which forces internal linkage. Otherwise, such functions default to external linkage, which might inhibit certain optimizations with some compilers—for example, aggressive inlining.

Dynamic Memory Allocation Consideration

Dynamic memory allocation (‘malloc’ in C language) should always return a pointer that is suitably aligned for the largest base type (quadword alignment). Where this aligned pointer cannot be guaranteed, use the technique shown in the following code to make the pointer quadword aligned, if needed. This code assumes the pointer can be cast to a long.

Example:

```
double* p;
double* np;

p  = (double *)malloc(sizeof(double)*number_of_doubles+7L);
np = (double *)((((long)(p))+7L) & (-8L));
```

Then use ‘np’ instead of ‘p’ to access the data. ‘p’ is still needed in order to deallocate the storage.

Introduce Explicit Parallelism into Code

Where possible, long dependency chains should be broken into several independent dependency chains which can then be executed in parallel exploiting the pipeline execution units. This is especially important for floating-point code, whether it is mapped to x87 or 3DNow! instructions because of the longer latency of floating-point operations. Since most languages, including ANSI C, guarantee that floating-point expressions are not re-ordered, compilers can not usually perform such optimizations unless they offer a switch to allow ANSI non-compliant reordering of floating-point expressions according to algebraic rules.

Note that re-ordered code that is algebraically identical to the original code does not necessarily deliver identical computational results due to the lack of associativity of floating point operations. There are well-known numerical considerations in applying these optimizations (consult a book on numerical analysis). In some cases, these optimizations may lead to unexpected results. Fortunately, in the vast majority of cases, the final result will differ only in the least significant bits.

Example 1 (Avoid):

```
double a[100],sum;
int i;

sum = 0.0f;
   for (i=0; i<100; i++) {
         sum += a[i];
   }
```

Example 2 (Preferred):

```
double a[100],sum1,sum2,sum3,sum4,sum;
int i;

sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
sum4 = 0.0;
for (i=0; i<100; i+4) {
   sum1 += a[i];
   sum2 += a[i+1];
   sum3 += a[i+2];
   sum4 += a[i+3];
   }
sum = (sum4+sum3)+(sum1+sum2);
```

Notice that the 4-way unrolling was chosen to exploit the 4-stage fully pipelined floating-point adder. Each stage of the floating-point adder is occupied on every clock cycle, ensuring maximal sustained utilization.

Explicitly Extract Common Subexpressions

In certain situations, C compilers are unable to extract common subexpressions from floating-point expressions due to the guarantee against reordering of such expressions in the ANSI standard. Specifically, the compiler can not re-arrange the computation according to algebraic equivalencies before extracting common subexpressions. In such cases, the programmer should manually extract the common subexpression. It should be noted that re-arranging the expression may result in different computational results due to the lack of associativity of floating-point operations, but the results usually differ in only the least significant bits.

Example 1

Avoid:

```
double  a,b,c,d,e,f;

e = b*c/d;
f = b/d*a;
```

Preferred:

```
double  a,b,c,d,e,f,t;

t = b/d;
e = c*t;
f = a*t;
```

Example 2

Avoid:

```
double a,b,c,e,f;

e = a/c;
f = b/c;
```

Preferred:

```
double a,b,c,e,f,t;

t = 1/c;
e = a*t
f = b*t;
```

C Language Structure Component Considerations

Many compilers have options that allow padding of structures to make their size multiples of words, doublewords, or quadwords, in order to achieve better alignment for structures. In addition, to improve the alignment of structure members, some compilers might allocate structure elements in an order that differs from the order in which they are declared. However, some compilers might not offer any of these features, or their implementation might not work properly in all situations. Therefore, to achieve the best alignment of structures and structure members while minimizing the amount of padding regardless of compiler optimizations, the following methods are suggested.

Sort by Base Type Size

Sort structure members according to their base type size, declaring members with a larger base type size ahead of members with a smaller base type size.

Pad by Multiple of Largest Base Type Size

Pad the structure to a multiple of the largest base type size of any member. In this fashion, if the first member of a structure is naturally aligned, all other members are naturally aligned as well. The padding of the structure to a multiple of the largest based type size allows, for example, arrays of structures to be perfectly aligned.

The following example demonstrates the reordering of structure member declarations:

Original ordering (Avoid):

```
struct {
       char    a[5];
       long    k;
       double  x;
} baz;
```

New ordering, with padding (Preferred):

```
struct {
       double  x;
       long    k;
       char    a[5];
       char    pad[7];
} baz;
```

See “C Language Structure Component Considerations” on page 47 for a different perspective.

Sort Local Variables According to Base Type Size

When a compiler allocates local variables in the same order in which they are declared in the source code, it can be helpful to declare local variables in such a manner that variables with a larger base type size are declared ahead of the variables with smaller base type size. Then, if the first variable is allocated so that it is naturally aligned, all other variables are allocated contiguously in the order they are declared, and are naturally aligned without any padding.

Some compilers do not allocate variables in the order they are declared. In these cases, the compiler should automatically allocate variables in such a manner as to make them naturally aligned with the minimum amount of padding. In addition, some compilers do not guarantee that the stack is aligned suitably for the largest base type (that is, they do not guarantee quadword alignment), so that quadword operands might be misaligned, even if this technique is used and the compiler does allocate variables in the order they are declared.

The following example demonstrates the reordering of local variable declarations:

Original ordering (Avoid):

```
short   ga, gu, gi;
long    foo, bar;
double  x, y, z[3];
char    a, b;
float   baz;
```

Improved ordering (Preferred):

```
double  z[3];
double  x, y;
long    foo, bar;
float   baz;
short   ga, gu, gi;
```

See “Sort Variables According to Base Type Size” on page 47 for more information from a different perspective.

Avoid Unnecessary Integer Division

Integer division is the slowest of all integer arithmetic operations and should be avoided wherever possible. One possibility for reducing the number of integer divisions is multiple divisions, in which division can be replaced with multiplication as shown in the following examples. This replacement is possible only if no overflow occurs during the computation of the product. This can be determined by considering the possible ranges of the divisors.

Example 1 (Avoid):

```
int i,j,k,m;

m = i / j / k;
```

Example 2 (Preferred):

```
int i,j,k,l;

m = i / (j * k);
```

Copy Frequently De-referenced Pointer Arguments to Local Variables

Avoid frequently de-referencing pointer arguments inside a function. Since the compiler has no knowledge of whether aliasing exists between the pointers, such de-referencing can not be optimized away by the compiler. This prevents data from being kept in registers and significantly increases memory traffic.

Note that many compilers have an “assume no aliasing” optimization switch. This allows the compiler to assume that two different pointers always have disjoint contents and does not require copying of pointer arguments to local variables.

Otherwise, copy the data pointed to by the pointer arguments to local variables at the start of the function and if necessary copy them back at the end of the function.

Example 1 (Avoid):

```
//assumes pointers are different
void isqrt  ( unsigned long a,
              unsigned long *q,
              unsigned long *r)
{
*q = a;
   if (a > 0)
   {
   while (*q > (*r = a / *q))
         {
         *q = (*q + *r) >> 1;
         }
   }
*r = a - *q * *q;
}
```

Example 2 (Preferred):

```
//assumes pointers are different
void isqrt  ( unsigned long a,
              unsigned long *q,
              unsigned long *r)
{
unsigned long qq, rr;
qq = a;
if (a > 0)
   {
   while (qq > (rr = a / qq))
         {
         qq = (qq + rr) >> 1;
         }
   }
rr = a - qq * qq;
*q = qq;
*r = rr;
}
```

4

Instruction Decoding Optimizations

This chapter discusses ways to maximize the number of instructions decoded by the instruction decoders in the AMD Athlon™ processor. Guidelines are listed in order of importance.

Overview

The AMD Athlon processor instruction fetcher reads 16-byte aligned code windows from the instruction cache. The instruction bytes are then merged into a 24-byte instruction queue. On each cycle, the in-order front-end engine selects for decode up to three x86 instructions from the instruction-byte queue.

All instructions (x86, x87, 3DNow!™, and MMX™) are classified into two types of decodes—DirectPath and VectorPath (see "DirectPath Decoder" and "VectorPath Decoder" on page 113 for more information). DirectPath instructions are common instructions that are decoded directly in hardware. VectorPath instructions are more complex instructions that require the use of a sequence of multiple operations issued from an on-chip ROM.

Up to three DirectPath instructions can be selected for decode per cycle. Only one VectorPath instruction can be selected for decode per cycle. DirectPath instructions and VectorPath instructions cannot be simultaneously decoded.

Select DirectPath Over VectorPath Instructions

Use DirectPath instructions rather than VectorPath instructions. DirectPath instructions are optimized for decode and execute efficiently by minimizing the number of operations per x86 instruction, which includes 'register←register op memory' as well as 'register←register op register' forms of instructions. Up to three DirectPath instructions can be decoded per cycle. VectorPath instructions may also block the decoding of DirectPath instructions. See Appendix D, "Instruction Dispatch and Execution Timing" on page 141 and Appendix E, "DirectPath versus VectorPath Instructions" on page 175 for tables of DirectPath and VectorPath instructions.

Load-Execute Instruction Usage

Use Load-Execute Integer Instructions

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

Use Load-Execute Floating-Point Instructions with Floating-Point Operands

When operating on single-precision or double-precision floating-point data, wherever possible use floating-point load-execute instructions to increase code density.

Note: *This optimization applies only to floating-point instructions with floating-point operands and not with integer operands, as described in the next optimization.*

This coding style helps in two ways. First, denser code allows more work to be held in the instruction cache. Second, the denser code generates fewer internal OPs and, therefore, the FPU scheduler holds more work, which increases the chances of extracting parallelism from the code.

Example 1 (Avoid):

```
FLD      QWORD PTR [TEST1]
FLD      QWORD PTR [TEST2]
FMUL     ST, ST(1)
```

Example 2 (Preferred):

```
FLD      QWORD PTR [TEST1]
FMUL     QWORD PTR [TEST2]
```

Avoid Load-Execute Floating-Point Instructions with Integer Operands

Do not use load-execute floating-point instructions with *integer* operands: FIADD, FISUB, FISUBR, FIMUL, FIDIV, FIDIVR, FICOM, and FICOMP. Remember that floating-point instructions can have integer operands while integer instruction cannot have floating-point operands.

Floating-point computations involving integer-memory operands should use separate FILD and arithmetic instructions. This optimization has the potential to increase decode bandwidth and OP density in the FPU scheduler. The floating-point load-execute instructions with integer operands are VectorPath and generate two OPs in a cycle, while the discrete equivalent enables a third DirectPath instruction to be decoded in the same cycle. In some situations this optimizations can also reduce execution time if the FILD can be scheduled several instructions ahead of the arithmetic instruction in order to cover the FILD latency.

Example 1 (Avoid):

```
FLD      QWORD PTR [foo]
FIMUL    DWORD PTR [bar]
FIADD    DWORD PTR [baz]
```

Example 2 (Preferred):

```
FILD     DWORD PTR [bar]
FILD     DWORD PTR [baz]
FLD      QWORD PTR [foo]
FMULP    ST(2), ST
FADDP    ST(1),ST
```

Align Branch Targets

Place branch targets at or near the beginning of 16-byte aligned code windows. This technique helps to maximize the number of instructions that are filled into the instruction-byte queue.

Use Short Instruction Lengths

Assemblers and compilers should generate the tightest code possible to optimize use of the I-Cache and increase average decode rate. Wherever possible, use instructions with shorter lengths. Using shorter instructions increases the number of instructions that can fit into the instruction-byte queue. For example, use 8-bit displacements as opposed to 32-bit displacements. In addition, use the single-byte format of simple integer instructions whenever possible, as opposed to the 2-byte Opcode ModR/M format.

Example 1 (Avoid):

```
81 C0 78 56 34 12   add eax, 12345678h  ;uses 2-byte opcode
                                        ; form (with ModR/M)
81 C3 FB FF FF FF   add ebx, -5         ;uses 32-bit
                                        ; immediate
0F 84 05 00 00 00   jz  $label1         ;uses 2-byte opcode,
                                        ; 32-bit immediate
```

Example 2 (Preferred):

```
05 78 56 34 12   add eax, 12345678h   ;uses single byte
                                      ; opcode form
83 C3 FB         add ebx, -5          ;uses 8-bit sign
                                      ; extended immediate
74 05            jz  $label1          ;uses 1-byte opcode,
                                      ; 8-bit immediate
```

Avoid Partial Register Reads and Writes

In order to handle partial register writes, the AMD Athlon processor execution core implements a data-merging scheme.

In the execution unit, an instruction writing a partial register merges the modified portion with the current state of the remainder of the register. Therefore, the dependency hardware can potentially force a false dependency on the most recent instruction that writes to any part of the register.

Example 1 (Avoid):

```
MOV      AL, 10      ;inst 1
MOV      AH, 12      ;inst 2 has a false dependency on
                     ; inst 1
                     ;inst 2 merges new AH with current
                     ; EAX register value forwarded
                     ; by inst 1
```

In addition, an instruction that has a read dependency on any part of a given architectural register has a read dependency on the most recent instruction that modifies any part of the same architectural register.

Example 2 (Avoid):

```
MOV      BX, 12h     ;inst 1
MOV      BL, DL      ;inst 2, false dependency on
                     ; completion of inst 1
MOV      BH, CL      ;inst 3, false dependency on
                     ; completion of inst 2
MOV      AL, BL      ;inst 4, depends on completion of
                     ; inst 2
```

Replace Certain SHLD Instructions with Alternative Code

Certain instances of the SHLD instruction can be replaced by alternative code using SHR and LEA. The alternative code has lower latency and requires less execution resources. SHR and LEA (32-bit version) are DirectPath instructions, while SHLD is a VectorPath instruction. SHR and LEA preserves decode bandwidth as it potentially enables the decoding of a third DirectPath instruction.

Example 1

(Avoid):

```
SHLD REG1, REG2, 1
```

(Preferred):

```
SHR REG2, 31
LEA REG1, [REG1*2 + REG2]
```

Example 2

(Avoid):

```
SHLD REG1, REG2, 2
```

(Preferred):

```
SHR REG2, 30
LEA REG1, [REG1*4 + REG2]
```

Example 3

(Avoid):

```
SHLD REG1, REG2, 3
```

(Preferred):

```
SHR REG2, 29
LEA REG1, [REG1*8 + REG2]
```

Use 8-Bit Sign-Extended Immediates

Using 8-bit sign-extended immediates improves code density with no negative effects on the AMD Athlon processor. For example, ADD BX, –5 should be encoded “83 C3 FB” and not “81 C3 FF FB”.

Use 8-Bit Sign-Extended Displacements

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 Athlon processor.

Code Padding Using Neutral Code Fillers

Occasionally a need arises to insert neutral code fillers into the code stream, e.g., for code alignment purposes or to space out branches. Since this filler code can be executed, it should take up as few execution resources as possible, not diminish decode density, and not modify any processor state other than advancing EIP. A one byte padding can easily be achieved using the NOP instructions (XCHG EAX,EAX; opcode 0x90). In the x86 architecture, there are several multi-byte "NOP" instructions available that do not change processor state other than EIP:

- MOV REG, REG
- XCHG REG, REG
- CMOVcc REG, REG
- SHR REG, 0
- SAR REG, 0
- SHL REG, 0
- SHRD REG, REG, 0
- SHLD REG, REG, 0
- LEA REG, [REG]
- LEA REG, [REG+00]
- LEA REG, [REG*1+00]
- LEA REG, [REG+00000000]
- LEA REG, [REG*1+00000000]

Not all of these instructions are equally suitable for purposes of code padding. For example, SHLD/SHRD are microcoded which reduces decode bandwidth and takes up execution resources.

Recommendations for the AMD Athlon™ Processor

For code that is optimized specifically for the AMD Athlon processor, the optimal code fillers are NOP instructions (opcode 0x90) with up to two REP prefixes (0xF3). In the AMD Athlon processor, a NOP with up to two REP prefixes can be handled by a single decoder with no overhead. As the REP prefixes are redundant and meaningless, they get discarded, and NOPs are handled without using any execution resources. The three

decoders of AMD Athlon processor can handle up to three NOPs, each with up to two REP prefixes each, in a single cycle, for a neutral code filler of up to nine bytes. If a larger amount of code padding is required, it is recommended to use a JMP instruction to jump across the padding region. The following assembly language macros show this:

```
NOP1_ATHLON TEXTEQU <DB 090h>
NOP2_ATHLON TEXTEQU <DB 0F3h, 090h>
NOP3_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h>
NOP4_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h, 090h>
NOP5_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h, 0F3h, 090h>
NOP6_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h, 0F3h, 0F3h, 090h>
NOP7_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h, 0F3h, 0F3h, 090h,
                     090h>
NOP8_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h, 0F3h, 0F3h, 090h,
                     0F3h, 090h>
NOP9_ATHLON TEXTEQU <DB 0F3h, 0F3h, 090h, 0F3h, 0F3h, 090h,
                     0F3h, 0F3h, 090h>
NOP10_ATHLONTEXTEQU <DB 0EBh, 008h, 90h, 90h, 90h, 90h,
                     90h,  90h, 90h, 90h>
```

Recommendations for AMD-K6® Family and AMD Athlon™ Processor Blended Code

On x86 processors other than the AMD Athlon processor (including the AMD-K6 family of processors), the REP prefix and especially multiple prefixes cause decoding overhead, so the above technique is not recommended for code that has to run well both on AMD Athlon processor **and** other x86 processors (blended code). In such cases the instructions and instruction sequences below are recommended. For neutral code fillers longer than eight bytes in length, the JMP instruction can be used to jump across the padding region.

Note that each of the instructions and instruction sequences below utilizes an x86 register. To avoid performance degradation, the register used in the padding should be selected so as to not lengthen existing dependency chains, i.e., one should select a register that is not used by instructions in the vicinity of the neutral code filler. Note that certain instructions use registers implicitly. For example, PUSH, POP, CALL, and RET all make implicit use of the ESP register. The 5-byte filler sequence below consists of two instructions. If flag changes across the code padding are acceptable, the following

instructions may be used as single instruction, 5-byte code fillers:

- TEST EAX, 0FFFF0000h
- CMP EAX, 0FFFF0000h

The following assembly language macros show the recommended neutral code fillers for code optimized for the AMD Athlon processor that also has to run well on other x86 processors. Note for some padding lengths, versions using ESP or EBP are missing due to the lack of fully generalized addressing modes.

```
NOP2_EAX TEXTEQU <DB 08Bh,0C0h> ;mov eax, eax
NOP2_EBX TEXTEQU <DB 08Bh,0DBh> ;mov ebx, ebx
NOP2_ECX TEXTEQU <DB 08Bh,0C9h> ;mov ecx, ecx
NOP2_EDX TEXTEQU <DB 08Bh,0D2h> ;mov edx, edx
NOP2_ESI TEXTEQU <DB 08Bh,0F6h> ;mov esi, esi
NOP2_EDI TEXTEQU <DB 08Bh,0FFh> ;mov edi, edi
NOP2_ESP TEXTEQU <DB 08Bh,0E4h> ;mov esp, esp
NOP2_EBP TEXTEQU <DB 08Bh,0EDh> ;mov ebp, ebp

NOP3_EAX TEXTEQU <DB 08Dh,004h,020h> ;lea eax, [eax]
NOP3_EBX TEXTEQU <DB 08Dh,01Ch,023h> ;lea ebx, [ebx]
NOP3_ECX TEXTEQU <DB 08Dh,00Ch,021h> ;lea ecx, [ecx]
NOP3_EDX TEXTEQU <DB 08Dh,014h,022h> ;lea edx, [edx]
NOP3_ESI TEXTEQU <DB 08Dh,024h,024h> ;lea esi, [esi]
NOP3_EDI TEXTEQU <DB 08Dh,034h,026h> ;lea edi, [edi]
NOP3_ESP TEXTEQU <DB 08Dh,03Ch,027h> ;lea esp, [esp]
NOP3_EBP TEXTEQU <DB 08Dh,06Dh,000h> ;lea ebp, [ebp]

NOP4_EAX TEXTEQU <DB 08Dh,044h,020h,000h> ;lea eax, [eax+00]
NOP4_EBX TEXTEQU <DB 08Dh,05Ch,023h,000h> ;lea ebx, [ebx+00]
NOP4_ECX TEXTEQU <DB 08Dh,04Ch,021h,000h> ;lea ecx, [ecx+00]
NOP4_EDX TEXTEQU <DB 08Dh,054h,022h,000h> ;lea edx, [edx+00]
NOP4_ESI TEXTEQU <DB 08Dh,064h,024h,000h> ;lea esi, [esi+00]
NOP4_EDI TEXTEQU <DB 08Dh,074h,026h,000h> ;lea edi, [edi+00]
NOP4_ESP TEXTEQU <DB 08Dh,07Ch,027h,000h> ;lea esp, [esp+00]

;lea eax, [eax+00];nop
NOP5_EAX TEXTEQU <DB 08Dh,044h,020h,000h,090h>

;lea ebx, [ebx+00];nop
NOP5_EBX TEXTEQU <DB 08Dh,05Ch,023h,000h,090h>

;lea ecx, [ecx+00];nop
NOP5_ECX TEXTEQU <DB 08Dh,04Ch,021h,000h,090h>

;lea edx, [edx+00];nop
NOP5_EDX TEXTEQU <DB 08Dh,054h,022h,000h,090h>
```

```
;lea esi, [esi+00];nop
NOP5_ESI TEXTEQU <DB 08Dh,064h,024h,000h,090h>

;lea edi, [edi+00];nop
NOP5_EDI TEXTEQU <DB 08Dh,074h,026h,000h,090h>

;lea esp, [esp+00];nop
NOP5_ESP TEXTEQU <DB 08Dh,07Ch,027h,000h,090h>

;lea eax, [eax+00000000]
NOP6_EAX TEXTEQU <DB 08Dh,080h,0,0,0,0>

;lea ebx, [ebx+00000000]
NOP6_EBX TEXTEQU <DB 08Dh,09Bh,0,0,0,0>

;lea ecx, [ecx+00000000]
NOP6_ECX TEXTEQU <DB 08Dh,089h,0,0,0,0>

;lea edx, [edx+00000000]
NOP6_EDX TEXTEQU <DB 08Dh,092h,0,0,0,0>

;lea esi, [esi+00000000]
NOP6_ESI TEXTEQU <DB 08Dh,0B6h,0,0,0,0>

;lea edi ,[edi+00000000]
NOP6_EDI TEXTEQU <DB 08Dh,0BFh,0,0,0,0>

;lea ebp ,[ebp+00000000]
NOP6_EBP TEXTEQU <DB 08Dh,0ADh,0,0,0,0>

;lea eax,[eax*1+00000000]
NOP7_EAX TEXTEQU <DB 08Dh,004h,005h,0,0,0,0>

;lea ebx,[ebx*1+00000000]
NOP7_EBX TEXTEQU <DB 08Dh,01Ch,01Dh,0,0,0,0>

;lea ecx,[ecx*1+00000000]
NOP7_ECX TEXTEQU <DB 08Dh,00Ch,00Dh,0,0,0,0>

;lea edx,[edx*1+00000000]
NOP7_EDX TEXTEQU <DB 08Dh,014h,015h,0,0,0,0>

;lea esi,[esi*1+00000000]
NOP7_ESI TEXTEQU <DB 08Dh,034h,035h,0,0,0,0>

;lea edi,[edi*1+00000000]
NOP7_EDI TEXTEQU <DB 08Dh,03Ch,03Dh,0,0,0,0>

;lea ebp,[ebp*1+00000000]
NOP7_EBP TEXTEQU <DB 08Dh,02Ch,02Dh,0,0,0,0>
```

```
;lea eax,[eax*1+00000000] ;nop
NOP8_EAX TEXTEQU <DB 08Dh,004h,005h,0,0,0,0,90h>

;lea ebx,[ebx*1+00000000] ;nop
NOP8_EBX TEXTEQU <DB 08Dh,01Ch,01Dh,0,0,0,0,90h>

;lea ecx,[ecx*1+00000000] ;nop
NOP8_ECX TEXTEQU <DB 08Dh,00Ch,00Dh,0,0,0,0,90h>

;lea edx,[edx*1+00000000] ;nop
NOP8_EDX TEXTEQU <DB 08Dh,014h,015h,0,0,0,0,90h>

;lea esi,[esi*1+00000000] ;nop
NOP8_ESI TEXTEQU <DB 08Dh,034h,035h,0,0,0,0,90h>

;lea edi,[edi*1+00000000] ;nop
NOP8_EDI TEXTEQU <DB 08Dh,03Ch,03Dh,0,0,0,0,90h>

;lea ebp,[ebp*1+00000000] ;nop
NOP8_EBP TEXTEQU <DB 08Dh,02Ch,02Dh,0,0,0,0,90h>

;JMP
NOP9 TEXTEQU <DB 0EBh,007h,90h,90h,90h,90h,90h,90h,90h>
```

5

Cache and Memory Optimizations

This chapter describes code optimization techniques that take advantage of the large L1 caches and high-bandwidth buses of the AMD Athlon™ processor. Guidelines are listed in order of importance.

Memory Size and Alignment Issues

Avoid Memory Size Mismatches

Avoid memory size mismatches when instructions operate on the same data. **For instructions that store and reload the same data, keep operands aligned and keep the loads/stores of each operand the same size.** The following code examples result in a store-to-load-forwarding (STLF) stall:

Example 1 (Avoid):

```
MOV   DWORD PTR [FOO], EAX
MOV   DWORD PTR [FOO+4], EDX
FLD   QWORD PTR [FOO]
```

Avoid large-to-small mismatches, as shown in the following code:

Example 2 (Avoid):

```
FST   QWORD PTR [FOO]
MOV   EAX, DWORD PTR [FOO]
MOV   EDX, DWORD PTR [FOO+4]
```

Align Data Where Possible

In general, avoid misaligned data references. All data whose size is a power of 2 is considered aligned if it is *naturally* aligned. For example:

- QWORD accesses are aligned if they access an address divisible by 8.
- DWORD accesses are aligned if they access an address divisible by 4.
- WORD accesses are aligned if they access an address divisible by 2.
- TBYTE accesses are aligned if they access an address divisible by 8.

A misaligned store or load operation suffers a minimum one-cycle penalty in the AMD Athlon processor load/store pipeline. In addition, using misaligned loads and stores increases the likelihood of encountering a store-to-load forwarding pitfall. For a more detailed discussion of store-to-load forwarding issues, see "Store-to-Load Forwarding Restrictions" on page 42.

Use the 3DNow!™ PREFETCH and PREFETCHW Instructions

For code that can take advantage of prefetching, use the 3DNow! PREFETCH and PREFETCHW instructions to increase the effective bandwidth to the AMD Athlon processor. The PREFETCH and PREFETCHW instructions take advantage of the AMD Athlon processor's high bus bandwidth to hide long latencies when fetching data from system memory. Large data sets typically require unit-stride access to ensure that all data pulled in by PREFETCH or PREFETCHW is actually used. If necessary, algorithms or data structures should be reorganized to allow unit-stride access.

PREFETCH/W versus PREFETCHNTA/T0/T1 /T2

The PREFETCHNTA/T0/T1/T2 instructions in the MMX extensions are processor implementation dependent. To maintain compatibility with the 25 million AMD-K6®-2 and AMD-K6-III processors already sold, use the 3DNow! PREFETCH/W instructions instead of the various prefetch flavors in the new MMX extensions.

PREFETCHW Usage

Code that intends to modify the cache line brought in through prefetching should use the PREFETCHW instruction. While PREFETCHW works the same as a PREFETCH on the AMD-K6-2 and AMD-K6-III processors, PREFETCHW gives a hint to the AMD Athlon processor of an intent to modify the cache line. The AMD Athlon processor will mark the cache line being brought in by PREFETCHW as Modified. Using PREFETCHW can save an additional 15-25 cycles compared to a PREFETCH and the subsequent cache state change caused by a write to the prefetched cache line.

Multiple Prefetches

Programmers can initiate multiple outstanding prefetches on the AMD Athlon processor. While the AMD-K6-2 and AMD-K6-III processors can have only one outstanding prefetch, the AMD Athlon processor can have up to six outstanding prefetches. For example, when traversing more than one array, the programmer should initiate multiple prefetches.

Example (Multiple Prefetches):

```
.CODE
.K3D

; original C code
;
; #define LARGE_NUM 65536
;
; double array_a[LARGE_NUM];
; double array b[LARGE_NUM];
; double array c[LARGE_NUM];
; int i;
;
; for (i = 0; i < LARGE_NUM; i++) {
;    a[i] = b[i] * c[i]
; }

MOV     ECX, (-LARGE_NUM)         ;used biased index
MOV     EAX, OFFSET array_a       ;get address of array_a
MOV     EDX, OFFSET array_b       ;get address of array_b
MOV     ECX, OFFSET array_c       ;get address of array_c

$loop:

PREFETCHW   [EAX+196]              ;two cachelines ahead
PREFETCH    [EDX+196]              ;two cachelines ahead
PREFETCH    [ECX+196]              ;two cachelines ahead
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE]   ;b[i]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE]   ;b[i]*c[i]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE]   ;a[i] = b[i]*c[i]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+8] ;b[i+1]
```

```
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+8] ;b[i+1]*c[i+1]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+8] ;a[i+1] =
                                       ; b[i+1]*c[i+1]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+16];b[i+2]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+16];b[i+2]*c[i+2]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+16];a[i+2] =
                                       ; [i+2]*c[i+2]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+24];b[i+3]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+24];b[i+3]*c[i+3]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+24];a[i+3] =
                                       ; b[i+3]*c[i+3]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+32];b[i+4]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+32];b[i+4]*c[i+4]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+32];a[i+4] =
                                       ; b[i+4]*c[i+4]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+40];b[i+5]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+40];b[i+5]*c[i+5]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+40];a[i+5] =
                                       ; b[i+5]*c[i+5]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+48];b[i+6]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+48];b[i+6]*c[i+6]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+48];a[i+6] =
                                       ; b[i+6]*c[i+6]
FLD   QWORD PTR [EDX+ECX*8+ARR_SIZE+56];b[i+7]
FMUL  QWORD PTR [ECX+ECX*8+ARR_SIZE+56];b[i+7]*c[i+7]
FSTP  QWORD PTR [EAX+ECX*8+ARR_SIZE+56];a[i+7] =
                                       ; b[i+7]*c[i+7]
ADD   ECX, 8                           ;next 8 products
JNZ   $loop                            ;until none left

END
```

The following optimization rules were applied to this example.

- Loops should be unrolled to make sure that the data stride per loop iteration is equal to the length of a cache line. This avoids overlapping PREFETCH instructions and thus optimal use of the available number of outstanding PREFETCHes.
- Since the array "array_a" is written rather than read, PREFETCHW is used instead of PREFETCH to avoid overhead for switching cache lines to the correct MESI state. The PREFETCH lookahead has been optimized such that each loop iteration is working on three cache lines while six active PREFETCHes bring in the next six cache lines.
- Index arithmetic has been reduced to a minimum by use of complex addressing modes and biasing of the array base addresses in order to cut down on loop overhead.

Determining Prefetch Distance

Given the latency of a typical AMD Athlon processor system and expected processor speeds, the following formula should be used to determine the prefetch distance in bytes:

Prefetch Distance = 200 ($^{DS}/_{C}$) bytes

- Round up to the nearest 64-byte cache line.
- The number 200 is a constant that is based upon expected AMD Athlon processor clock frequencies and typical system memory latencies.
- DS is the data stride in bytes per loop iteration.
- C is the number of cycles for one loop to execute entirely from the L1 cache.

Prefetch at Least 64 Bytes Away from Surrounding Stores

The PREFETCH and PREFETCHW instructions can be affected by false dependencies on stores. If there is a store to an address that matches a request, that request (the PREFETCH or PREFETCHW instruction) may be blocked until the store is written to the cache. Therefore, code should prefetch data that is located at **least** 64 bytes away from any surrounding store's data address.

Take Advantage of Write Combining

Operating system and device driver programmers should take advantage of the write-combining capabilities of the AMD Athlon processor. The AMD Athlon processor has a very aggressive write-combining algorithm, which improves performance significantly.

See Appendix C, "Implementation of Write Combining" on page 135 for more details.

Avoid Placing Code and Data in the Same 64-Byte Cache Line

Sharing code and data in the same 64-byte cache line may cause the L1 caches to thrash (unnecessary castout of code/data) in order to maintain coherency between the separate instruction and data caches. The AMD Athlon processor has a cache-line size of 64-bytes, which is twice the size of previous processors. Programmers must be aware that code and data should not be shared within this larger cache line, especially if the data becomes modified.

For example, programmers should consider that a memory indirect JMP instruction may have the data for the jump table residing in the same 64-byte cache line as the JMP instruction, which would result in lower performance.

Although rare, do not place critical code at the border between 32-byte aligned code segments and a data segments. The code at the start or end of your data segment should be as rarely executed as possible or simply padded with garbage.

In general, the following should be avoided:

- self-modifying code
- storing data in code segments

Store-to-Load Forwarding Restrictions

Store-to-load forwarding refers to the process of a load reading (forwarding) data from the store buffer (LS2). There are instances in the AMD Athlon processor load/store architecture when either a load operation is not allowed to read needed data from a store in the store buffer, or a load OP detects a false data dependency on a store in the store buffer.

In either case, the load cannot complete (load the needed data into a register) until the store has retired out of the store buffer and written to the data cache. A store-buffer entry cannot retire and write to the data cache until *every* instruction before the store has completed and retired from the reorder buffer.

The implication of this restriction is that all instructions in the reorder buffer, up to and including the store, must complete and retire out of the reorder buffer before the load can complete. Effectively, the load has a false dependency on every instruction up to the store.

The following sections discuss store-to-load forwarding examples that are acceptable and those that should be avoided.

Store-to-Load Forwarding Pitfalls—True Dependencies

A load is allowed to read data from the store-buffer entry only if all of the following conditions are satisfied:

- The start address of the load matches the start address of the store.
- The load operand size is equal to or smaller than the store operand size.
- Neither the load or store is misaligned.
- The store data is not from a high-byte register (AH, BH, CH, or DH).

The following sections describe common-case scenarios to avoid whereby a load has a true dependency on a LS2-buffered store but cannot read (forward) data from a store-buffer entry.

Narrow-to-Wide Store-Buffer Data Forwarding Restriction

If the following conditions are present, there is a narrow-to-wide store-buffer data forwarding restriction:

- The operand size of the store data is smaller than the operand size of the load data.
- The range of addresses spanned by the store data covers some sub-region of range of addresses spanned by the load data.

Avoid the type of code shown in the following two examples.

Example 1 (Avoid):

```
MOV EAX, 10h
MOV WORD PTR [EAX], BX      ;word store
...
MOV ECX, DWORD PTR [EAX]    ;doubleword load
                            ;cannot forward upper
                            ; byte from store buffer
```

Example 2 (Avoid):

```
MOV EAX, 10h
MOV BYTE PTR [EAX + 3], BL ;byte store
...
MOV ECX, DWORD PTR [EAX]   ;doubleword load
                           ;cannot forward upper byte
                           ; from store buffer
```

Wide-to-Narrow Store-Buffer Data Forwarding Restriction

If the following conditions are present, there is a wide-to-narrow store-buffer data forwarding restriction:

- The operand size of the store data is greater than the operand size of the load data.
- The start address of the store data does not match the start address of the load.

Example 3 (Avoid):

```
MOV EAX, 10h
ADD DWORD PTR [EAX], EBX   ;doubleword store
MOV CX, WORD PTR [EAX + 2] ;word load-cannot forward high
                           ; word from store buffer
```

Use Example 5 instead of Example 4.

Example 4 (Avoid):

```
MOVQ       [foo], MM1    ;store upper and lower half
...
ADD        EAX, [foo]    ;fine
ADD        EDX, [foo+4]  ;uh-oh!
```

Example 5 (Preferred):

```
MOVD       [foo], MM1    ;store lower half
PUNPCKHDQ  MM1, MM1      ;get upper half into lower half
MOVD       [foo+4], MM1  ;store lower half
...
ADD        EAX, [foo]    ;fine
ADD        EDX, [foo+4]  ;fine
```

Misaligned Store-Buffer Data Forwarding Restriction

If the following condition is present, there is a misaligned store-buffer data forwarding restriction:

- The store or load address is misaligned. For example, a quadword store is not aligned to a quadword boundary, a doubleword store is not aligned to doubleword boundary, etc.

A common case of misaligned store-data forwarding involves the passing of misaligned quadword floating-point data on the

doubleword-aligned integer stack. Avoid the type of code shown in the following example.

Example 6 (Avoid):

```
MOV   ESP, 24h
FSTP  QWORD PTR [ESP]  ;esp=24
.                      ;store occurs to quadword
.                      ; misaligned address
.
FLD   QWORD PTR[ESP]   ;quadword load cannot forward
                       ; from quadword misaligned
                       ; 'fstp[esp]' store OP
```

High-Byte Store-Buffer Data Forwarding Restriction

If the following condition is present, there is a high-byte store-data buffer forwarding restriction:

- The store data is from a high-byte register (AH, BH, CH, DH).

Avoid the type of code shown in the following example.

Example 7 (Avoid):

```
MOV EAX, 10h
MOV [EAX], BH        ;high-byte store
.
MOV DL, [EAX]        ;load cannot forward from
                     ; high-byte store
```

One Supported Store-to-Load Forwarding Case

There is one case of a mismatched store-to-load forwarding that is supported by the by AMD Athlon processor. The lower 32 bits from an aligned QWORD write feeding into a DWORD read is allowed.

Example 8 (Allowed):

```
MOVQ   [AlignedQword], mm0
...
MOV    EAX, [AlignedQword]
```

Summary of Store-to-Load Forwarding Pitfalls to Avoid

To avoid store-to-load forwarding pitfalls, code should conform to the following guidelines:

- Maintain consistent use of operand size across all loads and stores. Preferably, use doubleword or quadword operand sizes.
- Avoid misaligned data references.
- Avoid narrow-to-wide and wide-to-narrow forwarding cases.

- When using word or byte stores, avoid loading data from anywhere in the same doubleword of memory other than the identical start addresses of the stores.

Stack Alignment Considerations

Make sure the stack is suitably aligned for the local variable with the largest base type. Then, using the technique described in "C Language Structure Component Considerations" on page 47, all variables can be properly aligned with no padding.

Extend to 32 Bits Before Pushing onto Stack

Function arguments smaller than 32 bits should be extended to 32 bits before being pushed onto the stack, which ensures that the stack is always doubleword aligned on entry to a function.

If a function has no local variables with a base type larger than doubleword, no further work is necessary. If the function does have local variables whose base type is larger than a doubleword, additional code should be inserted to ensure proper alignment of the stack. For example, the following code achieves quadword alignment:

Example (Preferred):

```
Prolog:
PUSH    EBP
MOV     EBP, ESP
AND     ESP, -8
SUB     ESP, SIZE_OF_LOCALS     ;size of local variables
                ;push registers that need to be preserved

Epilog:         ;pop register that needed to be preserved
MOV     ESP, EBP
POP     EBP
RET
```

With this technique, function arguments can be accessed via EBP, and local variables can be accessed via ESP. In order to free EBP for general use, it needs to be saved and restored between the prolog and the epilog.

Align TBYTE Variables on Quadword Aligned Addresses

Align variables of type TBYTE on quadword aligned addresses. In order to make an array of TBYTE variables that are aligned, array elements are 16-bytes apart. In general, TBYTE variables should be avoided. Use double-precision variables instead.

C Language Structure Component Considerations

Structures ('struct' in C language) should be made the size of a multiple of the largest base type of any of their components. To meet this requirement, padding should be used where necessary.

Language definitions permitting, to minimize padding, structure components should be sorted and allocated such that the components with a larger base type are allocated ahead of those with a smaller base type. For example, consider the following code:

Example:

```
struct   {
   char  a[5];
   long  k;
   doublex;
   } baz;
```

The structure components should be allocated (lowest to highest address) as follows:

```
x, k, a[4], a[3], a[2], a[1], a[0], padbyte6, ..., padbyte0
```

See "C Language Structure Component Considerations" on page 21 for more information from a C source code perspective.

Sort Variables According to Base Type Size

Sort local variables according to their base type size and allocate variables with larger base type size ahead of those with smaller base type size. Assuming the first variable allocated is naturally aligned, all other variables are naturally aligned

without any padding. The following example is a declaration of local variables in a C function:

Example:

```
short    ga, gu, gi;
long     foo, bar;
double   x, y, z[3];
char     a, b;
float    baz;
```

Allocate in the following order from left to right (from higher to lower addresses):

```
x, y, z[2], z[1], z[0], foo, bar, baz, ga, gu, gi, a, b;
```

See “Sort Local Variables According to Base Type Size” on page 22 for more information from a C source code perspective.

6

Branch Optimizations

While the AMD Athlon™ processor contains a very sophisticated branch unit, certain optimizations increase the effectiveness of the branch prediction unit. This chapter discusses rules that improve branch prediction and minimize branch penalties. Guidelines are listed in order of importance.

Avoid Branches Dependent on Random Data

Avoid conditional branches depending on random data, as these are difficult to predict. For example, a piece of code receives a random stream of characters “A” through “Z” and branches if the character is before “M” in the collating sequence. Data-dependent branches acting upon basically random data causes the branch prediction logic to mispredict the branch about 50% of the time.

If possible, design branch-free alternative code sequences, which results in shorter average execution time. This technique is especially important if the branch body is small. Examples 1 and 2 illustrate this concept using the CMOV instruction. Note that the AMD-K6® processor does not support the CMOV instruction. Therefore, blended AMD-K6 and AMD Athlon processor code should use Examples 3 and 4.

AMD Athlon™ Processor Specific Code

Example 1 – Signed integer ABS function (X = labs(X)):

```
MOV      ECX, [X]      ;load value
MOV      EBX, ECX      ;save value
NEG      ECX           ;-value
CMOVS    ECX, EBX      ;if -value is negative, select value
MOV      [X], ECX      ;save labs result
```

Example 2 – Unsigned integer min function (z = x < y ? x : y):

```
MOV      EAX, [X]         ;load X value
MOV      EBX, [Y]         ;load Y value
CMP      EAX, EBX         ;EBX<=EAX ? CF=0 : CF=1
CMOVNC   EAX, EBX         ;EAX=(EBX<=EAX) ? EBX:EAX
MOV      [Z], EAX         ;save min (X,Y)
```

Blended AMD-K6® and AMD Athlon™ Processor Code

Example 3 – Signed integer ABS function (X = labs(X)):

```
MOV    ECX, [X]    ;load value
MOV    EBX, ECX    ;save value
SAR    ECX, 31     ;x < 0 ? 0xffffffff : 0
XOR    EBX, ECX    ;x < 0 ? ~x : x
SUB    EBX, ECX    ;x < 0 ? (~x)+1 : x
MOV    [X], EBX    ;x < 0 ? -x : x
```

Example 4 – Unsigned integer min function (z = x < y ? x : y):

```
MOV    EAX, [x]    ;load x
MOV    EBX, [y]    ;load y
SUB    EAX, EBX    ;x < y ? CF : NC ; x - y
SBB    ECX, ECX    ;x < y ? 0xffffffff : 0
AND    ECX, EAX    ;x < y ? x - y : 0
ADD    ECX, EBX    ;x < y ? x - y + y : y
MOV    [z], ECX    ;x < y ? x : y
```

Example 5 – Hexadecimal to ASCII conversion (y=x < 10 ? x + 0x30: x + 0x41):

```
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: Sub. 60h
MOV      [Y],AL      ;save conversion in y
```

Example 6 – Increment Ring Buffer Offset:

```
//C Code
char buf[BUFSIZE];
   int a;

   if (a < (BUFSIZE-1)) {
      a++;
   } else {
      a = 0;
   }

;-------------
;Assembly Code
MOV     EAX, [a]          ; old offset
CMP     EAX, (BUFSIZE-1)  ; a < (BUFSIZE-1) ? CF : NC
INC     EAX               ; a++
SBB     EDX, EDX          ; a < (BUFSIZE-1) ? 0xffffffff :0
AND     EAX, EDX          ; a < (BUFSIZE-1) ? a++ : 0
MOV     [a], EAX          ; store new offset
```

Example 7 – Integer Signum Function:

```
//C Code
int a, s;

if (!a) {
      s = 0;
} else if (a < 0) {
      s = -1;
} else {
      s = 1;
}

;-------------
;Assembly Code
MOV     EAX, [a]         ;load a
CDQ                      ;t = a < 0 ? 0xffffffff : 0
CMP     EDX, EAX         ;a > 0 ? CF : NC
ADC     EDX, 0           ;a > 0 ? t+1 : t
MOV     [s], EDX         ;signum(x)
```

Always Pair CALL and RETURN

When the 12 entry return address stack gets out of synchronization, the latency of returns increase. The return address stack becomes out of sync when:

- calls and returns do not match
- the depth of the return stack is exceeded because of too many levels of nested functions calls

Replace Branches with Computation in 3DNow!™ Code

Branches negatively impact the performance of 3DNow! code. Branches can operate only on one data item at a time, i.e., they are inherently scalar and inhibit the SIMD processing that makes 3DNow! code superior. Also, branches based on 3DNow! comparisons require data to be passed to the integer units, which requires either transport through memory, or the use of "MOVD reg, MMreg" instructions. If the body of the branch is small, one can achieve higher performance by replacing the branch with computation. The computation simulates predicated execution or conditional moves. The principal tools for this are the following instructions: PCMPGT, PFCMPGT, PFCMPGE, PFMIN, PFMAX, PAND, PANDN, POR, PXOR.

Muxing Constructs

The most important construct to avoiding branches in 3DNow!™ and MMX™ code is a 2-way muxing construct that is equivalent to the ternary operator "?:" in C and C++. It is implemented using the PCMP/PFCMP, PAND, PANDN, and POR instructions. To maximize performance, it is important to apply the PAND and PANDN instructions in the proper order.

Example 1 (Avoid):

```
; r = (x < y) ? a : b
;
; in:  mm0  a
;      mm1  b
;      mm2  x
;      mm3  y
; out: mm1  r

PCMPGTD  MM3, MM2   ; y > x ? 0xffffffff : 0
MOVQ     MM4, MM3   ; duplicate mask
PANDN    MM3, MM0   ; y > x ? 0 : a
PAND     MM1, MM4   ; y > x ? b : 0
POR      MM1, MM3   ; r = y > x ? b : a
```

Because the use of PANDN destroys the mask created by PCMP, the mask needs to be saved, which requires an additional register. This adds an instruction, lengthens the dependency chain, and increases register pressure. Therefore 2-way muxing constructs should be written as follows.

Example 2 (Preferred):

```
; r = (x < y) ? a : b
;
; in:  mm0  a
;      mm1  b
;      mm2  x
;      mm3  y
; out: mm1  r

PCMPGTD  MM3, MM2   ; y > x ? 0xffffffff : 0
PAND     MM1, MM3   ; y > x ? b : 0
PANDN    MM3, MM0   ; y > x > 0 : a
POR      MM1, MM3   ; r = y > x ? b : a   "
```

Sample Code Translated Into 3DNow!™ Code

The following examples use scalar code translated into 3DNow! code. Note that it is not recommended to use 3DNow! SIMD instructions for scalar code, because the advantage of 3DNow! instructions lies in their "SIMDness". These examples are meant to demonstrate general techniques for translating source code with branches into branchless 3DNow! code. Scalar source code was chosen to keep the examples simple. These techniques work in an identical fashion for vector code.

Each example shows the C code and the resulting 3DNow! code.

Example 1:

C code:

```
float x,y,z;
if (x < y) {
   z += 1.0;
}
else {
   z -= 1.0;
}
```

3DNow! code:

```
;in:  MM0 = x
;     MM1 = y
;     MM2 = z
;out: MM0 = z
MOVQ     MM3, MM0    ;save x
MOVQ     MM4, one    ;1.0
PFCMPGE  MM0, MM1    ;x < y ? 0 : 0xffffffff
PSLLD    MM0, 31     ;x < y ? 0 : 0x80000000
PXOR     MM0, MM4    ;x < y ? 1.0 : -1.0
PFADD    MM0, MM2    ;x < y ? z+1.0 : z-1.0
```

Example 2:

C code:

```
float x,z;
z = abs(x);
if (z >= 1) {
   z = 1/z;
}
```

3DNow! code:

```
;in:  MM0 = x
;out: MM0 = z
MOVQ     MM5, mabs  ;0x7fffffff
PAND     MM0, MM5   ;z=abs(x)
PFRCP    MM2, MM0   ;1/z approx
MOVQ     MM1, MM0   ;save z
PFRCPIT1 MM0, MM2   ;1/z step
PFRCPIT2 MM0, MM2   ;1/z final
PFMIN    MM0, MM1   ;z = z < 1 ? z : 1/z
```

Example 3:

C code:

```
float x,z,r,res;
z = fabs(x)
if (z < 0.575) {
     res = r;
}
else {
     res = PI/2 - 2*r;
}
```

3DNow! code:

```
;in:  MM0 = x
;     MM1 = r
;out: MM0 = res
MOVQ    MM7, mabs ;mask for absolute value
PAND    MM0, MM7  ;z = abs(x)
MOVQ    MM2, bnd  ;0.575
PCMPGTD MM2, MM0  ;z < 0.575 ? 0xffffffff : 0
MOVQ    MM3, pio2 ;pi/2
MOVQ    MM0, MM1  ;save r
PFADD   MM1, MM1  ;2*r
PFSUBR  MM1, MM3  ;pi/2 - 2*r
PAND    MM0, MM2  ;z < 0.575 ? r : 0
PANDN   MM2, MM1  ;z < 0.575 ? 0 : pi/2 - 2*r
POR     MM0, MM2  ;z < 0.575 ? r : pi/2 - 2 * r
```

Example 4:

C code:

```
#define PI 3.14159265358979323
float x,z,r,res;
/* 0 <= r <= PI/4 */
z = abs(x)
if (z < 1) {
     res = r;
}
else {
     res = PI/2-r;
}
```

3DNow! code:

```
;in:  MM0 = x
;     MM1 = r
;out: MM1 = res
MOVQ     MM5, mabs  ; mask to clear sign bit
MOVQ     MM6, one   ; 1.0
PAND     MM0, MM5   ; z=abs(x)
PCMPGTD  MM6, MM0   ; z < 1 ? 0xffffffff : 0
MOVQ     MM4, pio2  ; pi/2
PFSUB    MM4, MM1   ; pi/2-r
PANDN    MM6, MM4   ; z < 1 ? 0 : pi/2-r
PFMAX    MM1, MM6   ; res = z < 1 ? r : pi/2-r
```

Example 5:

C code:

```
#define PI 3.14159265358979323
float x,y,xa,ya,r,res;
int   xs,df;
xs = x < 0 ? 1 : 0;
xa = fabs(x);
ya = fabs(y);
df = (xa < ya);
if (xs && df) {
     res = PI/2 + r;
}
else if (xs) {
     res = PI - r;
}
else if (df) {
     res = PI/2 - r;
}
else {
     res = r;
}
```

3DNow! code:

```
;in:  MM0 = r
;     MM1 = y
;     MM2 = x
;out: MM0 = res
MOVQ     MM7, sgn     ;mask to extract sign bit
MOVQ     MM6, sgn     ;mask to extract sign bit
MOVQ     MM5, mabs    ;mask to clear sign bit
PAND     MM7, MM2     ;xs = sign(x)
PAND     MM1, MM5     ;ya = abs(y)
PAND     MM2, MM5     ;xa = abs(x)
MOVQ     MM6, MM1     ;y
PCMPGTD  MM6, MM2     ;df = (xa < ya) ? 0xffffffff : 0
PSLLD    MM6, 31      ;df = bit<31>
MOVQ     MM5, MM7     ;xs
PXOR     MM7, MM6     ;xs^df ? 0x80000000 : 0
MOVQ     MM3, npio2   ;-pi/2
PXOR     MM5, MM3     ;xs ? pi/2 : -pi/2
PSRAD    MM6, 31      ;df ? 0xffffffff : 0
PANDN    MM6, MM5     ;xs ? (df ? 0 : pi/2) : (df ? 0 : -pi/2)
PFSUB    MM6, MM3     ;pr = pi/2 + (xs ? (df ? 0 : pi/2) :
                      ; (df ? 0 : -pi/2))
POR      MM0, MM7     ;ar = xs^df ? -r : r
PFADD    MM0, MM6     ;res = ar + pr
```

Avoid the Loop Instruction

The LOOP instruction in the AMD Athlon processor requires eight cycles to execute. Use the preferred code shown below:

Example 1 (Avoid):

```
LOOP     LABEL
```

Example 2 (Preferred):

```
DEC      ECX
JNZ      LABEL
```

Avoid Far Control Transfer Instructions

Avoid using far control transfer instructions. Far control transfer branches can not be predicted by the branch target buffer (BTB).

Avoid Recursive Functions

Avoid recursive functions due to the danger of overflowing the return address stack. Convert end-recursive functions to iterative code. An end-recursive function is when the function call to itself is at the end of the code.

Example 1 (Avoid):

```
long fac(long a)
{
   if (a==0) {
         return (1);
   } else {
         return (a*fac(a-1));
   }
   return (t);
}
```

Example 2 (Preferred):

```
long fac(long a)
{
   long t=1;
   while (a > 0) {
         t *= a;
         a--;
   }
   return (t);
}
```

7

Scheduling Optimizations

This chapter describes how to code instructions for efficient scheduling. Guidelines are listed in order of importance.

Schedule Instructions According to Their Latency

The AMD Athlon™ processor can execute up to three x86 instructions per cycle, with each x86 instruction possibly having a different latency. The AMD Athlon processor has flexible scheduling, but for absolute maximum performance, schedule instructions, especially FPU and 3DNow! instructions, according to their latency. Dependent instructions will then not have to wait on instructions with longer latencies.

See Appendix D, “Instruction Dispatch and Execution Timing” on page 141 for a list of latency numbers.

Unrolling Loops

Complete Loop Unrolling

Make use of the large AMD Athlon processor 64-Kbyte instruction cache and unroll loops to get more parallelism and reduce loop overhead, even with branch prediction. Complete

unrolling reduces register pressure by removing the loop counter. To completely unroll a loop, remove the loop control and replicate the loop body N times. In addition, completely unrolling a loop increases scheduling opportunities.

Only unrolling very large code loops can result in the inefficient use of the L1 instruction cache. Loops can be unrolled completely, if all of the following conditions are true:

- The loop is in a frequently executed piece of code.
- The loop count is known at compile time.
- The loop body, once unrolled, is less than 100 instructions, which is approximately 400 bytes of code.

Partial Loop Unrolling

Partial loop unrolling can increase register pressure, which can make it inefficient due to the small number of registers in the x86 architecture. However, in certain situations, partial unrolling can be efficient due to the performance gains possible. Partial loop unrolling should be considered if the following conditions are met:

- Spare registers are available
- The loop body is small, so that loop overhead is significant
- The number of loop iterations is likely > 10

Consider the following piece of C code:

```
double a[MAX_LENGTH], b[MAX_LENGTH];

   for (i=0; i< MAX_LENGTH; i++) {
      a[i] = a[i] + b[i];
   }
```

Without loop unrolling, the code looks like the following:

Without Loop Unrolling:

```
MOV ECX, MAX_LENGTH
MOV EAX, OFFSET A
MOV EBX, OFFSET B

$add_loop:
FLD      QWORD PTR [EAX]
FADD     QWORD PTR [EBX]
FSTP     QWORD PTR [EAX]
ADD      EAX, 8
ADD      EBX, 8
DEC      ECX
JNZ      $add_loop
```

The loop consists of seven instructions. The AMD Athlon processor can decode/retire three instructions per cycle, so it cannot execute faster than three iterations in seven cycles, or 3/7 floating-point adds per cycle. However, the pipelined floating-point adder allows one add every cycle. In the following code, the loop is partially unrolled by a factor of two, which creates potential endcases that must be handled outside the loop:

With Partial Loop Unrolling:

```
MOV      ECX, MAX_LENGTH
MOV      EAX, offset A
MOV      EBX, offset B
SHR      ECX, 1
JNC      $add_loop
FLD      QWORD PTR [EAX]
FADD     QWORD PTR [EBX]
FSTP     QWORD PTR [EAX]
ADD      EAX, 8
ADD      EBX, 8

$add_loop:
FLD      QWORD PTR[EAX]
FADD     QWORD PTR[EBX]
FSTP     QWORD PTR[EAX]
FLD      QWORD PTR[EAX+8]
FADD     QWORD PTR[EBX+8]
FSTP     QWORD PTR[EAX+8]
ADD      EAX, 16
ADD      EBX, 16
DEC      ECX
JNZ      $add_loop
```

Now the loop consists of 10 instructions. Based on the decode/retire bandwidth of three OPs per cycle, this loop goes

no faster than three iterations in 10 cycles, or 6/10 floating-point adds per cycle, or 1.4 times as fast as the original loop.

Deriving Loop Control For Partially Unrolled Loops

A frequently used loop construct is a counting loop. In a typical case, the loop count starts at some lower bound `lo`, increases by some fixed, positive increment `inc` for each iteration of the loop, and may not exceed some upper bound `hi`. The following example shows how to partially unroll such a loop by an unrolling factor of `fac`, and how to derive the loop control for the partially unrolled version of the loop.

Example 1 (rolled loop):

```
for (k = lo; k <= hi; k += inc) {
   x[k] =
   ...
}
```

Example 2 (partially unrolled loop):

```
for (k = lo; k <= (hi - (fac-1)*inc); k += fac*inc) {
   x[k] =
   ...
   x[k+inc] =
   ...
   ...
   x[k+(fac-1)*inc] =
   ...
}

/* handle end cases */

for (k = k; k <= hi; k += inc) {
   x[k] =
   ...
}
```

Use Function Inlining

Overview

Make use of the AMD Athlon processor's large 64-Kbyte instruction cache by inlining small routines to avoid procedure-call overhead. Consider the cost of possible increased register usage, which can increase load/store instructions for register spilling.

Function inlining has the advantage of eliminating function call overhead and allowing better register allocation and instruction scheduling at the site of the function call. The disadvantage is decreasing code locality, which can increase execution time due to instruction cache misses. Therefore, function inlining is an optimization that has to be used judiciously.

In general, due to its very large instruction cache, the AMD Athlon processor is less susceptible than other processors to the negative side effect of function inlining. Function call overhead on the AMD Athlon processor can be low because calls and returns are executed at high speed due to the use of prediction mechanisms. However, there is still overhead due to passing function arguments through memory, which creates STLF (store-to-load-forwarding) dependencies. Some compilers allow for a reduction of this overhead by allowing arguments to be passed in registers in one of their calling conventions, which has the drawback of constraining register allocation in the function and at the site of the function call.

In general, function inlining works best if the compiler can utilize feedback from a profiler to identify the function call sites most frequently executed. If such data is not available, a reasonable heuristic is to concentrate on function calls inside loops. Functions that are directly recursive should not be considered candidates for inlining. However, if they are end-recursive, the compiler should convert them to an iterative equivalent to avoid potential overflow of the AMD Athlon processor return prediction mechanism (return stack) during deep recursion. For best results, a compiler should support function inlining across multiple source files. In addition, a compiler should provide inline templates for commonly used library functions, such as sin(), strcmp(), or memcpy().

Always Inline Functions If Called From One Site

A function should always be inlined if it can be established that it is called from just one site in the code. For the C language, determination of this characteristic is made easier if functions are explicitly declared static unless they require external linkage. This case occurs quite frequently, as functionality that could be concentrated in a single large function is split across

multiple small functions for improved maintainability and readability.

Always Inline Functions with Fewer Than 25 Machine Instructions

In addition, functions that create fewer than 25 machine instructions once inlined should always be inlined because it is likely that the function call overhead is close to or more than the time spent executing the function body. For large functions, the benefits of reduced function call overhead gives diminishing returns. Therefore, a function that results in the insertion of more than 500 machine instructions at the call site should probably not be inlined. Some larger functions might consist of multiple, relatively short paths that are negatively affected by function overhead. In such a case, it can be advantageous to inline larger functions. Profiling information is the best guide in determining whether to inline such large functions.

Avoid Address Generation Interlocks

Loads and stores are scheduled by the AMD Athlon processor to access the data cache in program order. Newer loads and stores with their addresses calculated can be blocked by older loads and stores whose addresses are not yet calculated – this is known as an address generation interlock. Therefore, it is advantageous to schedule loads and stores that can calculate their addresses quickly, ahead of loads and stores that require the resolution of a long dependency chain in order to generate their addresses. Consider the following code examples.

Example 1 (Avoid):

```
ADD EBX, ECX                  ;inst 1
MOV EAX, DWORD PTR [10h]      ;inst 2 (fast address calc.)
MOV ECX, DWORD PTR [EAX+EBX]  ;inst 3 (slow address calc.)
MOV EDX, DWORD PTR [24h]      ;this load is stalled from
                              ; accessing data cache due
                              ; to long latency for
                              ; generating address for
                              ; inst 3
```

Example 2 (Preferred):

```
ADD EBX, ECX                    ;inst 1
MOV EAX, DWORD PTR [10h]        ;inst 2
MOV EDX, DWORD PTR [24h]        ;place load above inst 3
                                ; to avoid address
                                ; generation interlock stall
MOV ECX, DWORD PTR [EAX+EBX]    ;inst 3
```

Use MOVZX and MOVSX

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

Example 1 (Avoid):

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

Example 2 (Preferred):

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

Minimize Pointer Arithmetic in Loops

Minimize pointer arithmetic in loops, especially if the loop body is small. In this case, the pointer arithmetic would cause significant overhead. Instead, take advantage of the complex addressing modes to utilize the loop counter to index into memory arrays. Using complex addressing modes does not have any negative impact on execution speed, but the reduced number of instructions preserves decode bandwidth.

Example 1 (Avoid):

```
int a[MAXSIZE], b[MAXSIZE], c[MAXSIZE], i;

for (i=0; i < MAXSIZE; i++) {
       c [i] = a[i] + b[i];
}

MOV    ECX, MAXSIZE   ;initialize loop counter
XOR    ESI, ESI       ;initialize offset into array a
XOR    EDI, EDI       ;initialize offset into array b
XOR    EBX, EBX       ;initialize offset into array c
```

```
$add_loop:
MOV    EAX, [ESI + a] ;get element a
MOV    EDX, [EDI + b] ;get element b
ADD    EAX, EDX       ;a[i] + b[i]
MOV    [EBX + c], EAX ;write result to c
ADD    ESI, 4         ;increment offset into a
ADD    EDI, 4         ;increment offset into b
ADD    EBX, 4         ;increment offset into c
DEC    ECX            ;decrement loop count
JNZ    $add_loop      ;until loop count 0
```

Example 2 (Preferred):

```
int a[MAXSIZE], b[MAXSIZE], c[MAXSIZE], i;

for (i=0; i < MAXSIZE; i++) {
       c [i] = a[i] + b[i];
}

MOV  ECX, MAXSIZE-1    ;initialize loop counter

$add_loop:
MOV  EAX, [ECX*4 + a]  ;get element a
MOV  EDX, [ECX*4 + b]  ;get element b
ADD  EAX, EDX          ;a[i] + b[i]
MOV  [ECX*4 + c], EAX  ;write result to c
DEC  ECX               ;decrement index
JNS  $add_loop         ;until index negative
```

Note that the code in example 2 traverses the arrays in a downward direction (i.e., from higher addresses to lower addresses), whereas the original code in example 1 traverses the arrays in an upward direction. Such a change in the direction of the traversal is possible if each loop iteration is completely independent of all other loop iterations, as is the case here.

In code where the direction of the array traversal can't be switched, it is still possible to minimize pointer arithmetic by appropriately biasing base addresses and using an index variable that starts with a negative value and reaches zero when the loop expires. Note that if the base addresses are held in registers (e.g., when the base addresses are passed as arguments of a function) biasing the base addresses requires additional instructions to perform the biasing at run time and a small amount of additional overhead is incurred. In the examples shown here the base addresses are used in the displacement portion of the address and biasing is

accomplished at compile time by simply modifying the displacement.

Example 3 (Preferred):

```
int a[MAXSIZE], b[MAXSIZE], c[MAXSIZE], i;

for (i=0; i < MAXSIZE; i++) {
   c [i] = a[i] + b[i];
}

MOV   ECX, (-MAXSIZE)              ;initialize index

$add_loop:
MOV   EAX, [ECX*4 + a + MAXSIZE*4] ;get a element
MOV   EDX, [ECX*4 + b + MAXSIZE*4] ;get b element
ADD   EAX, EDX                     ;a[i] + b[i]
MOV   [ECX*4 + c + MAXSIZE*4], EAX ;write result to c
INC   ECX                          ;increment index
JNZ   $add_loop                    ;until index==0
```

Push Memory Data Carefully

Carefully choose the best method for pushing memory data. To reduce register pressure and code dependencies, follow example 2 below.

Example 1 (Avoid):

```
MOV   EAX, [MEM]
PUSH  EAX
```

Example 2 (Preferred):

```
PUSH  [MEM]
```

8

Integer Optimizations

This chapter describes ways to improve integer performance through optimized programming techniques. The guidelines are listed in order of importance.

Replace Divides with Multiplies

Replace integer division by constants with multiplication by the reciprocal. Because the AMD Athlon™ processor has a very fast integer multiply (5–9 cycles signed, 4–8 cycles unsigned) and the integer division delivers only one bit of quotient per cycle (22–47 cycles signed, 17–41 cycles unsigned), the equivalent code is much faster. The user can follow the examples in this chapter that illustrate the use of integer division by constants, or access the executables in the opt_utilities directory in the AMD documentation CD-ROM (PID 21860) to find alternative code for dividing by a constant.

Multiplication by Reciprocal (Division) Utility

The code for the utilities can be found at "Derivation of Multiplier Used For Integer Division by Constants" on page 84. All utilities were compiled for the Microsoft Windows® 95, Windows 98, and Windows NT® environments. All utilities are provided 'as is' and are not supported by AMD.

Signed Division Utility

In the opt_utilities directory of the AMD documentation CDROM, run **sdiv.exe** in a DOS box to find the fastest code for *signed* division by a constant. The utility displays the code after the user enters a signed constant divisor. Type "sdiv > example.out" to output the code to a file.

Unsigned Division Utility

In the opt_utilities directory of the AMD documentation CDROM, run **udiv.exe** in a DOS box to find the fastest code for *unsigned* division by a constant. The utility displays the code after the user enters an unsigned constant divisor. Type "udiv > example.out" to output the code to a file.

Unsigned Division by Multiplication of Constant

Algorithm: Divisors $1 <= d < 2^{31}$, Odd d

The following code shows an unsigned division using a constant value multiplier.

```
;In:  d = divisor, 1 <= d < 2^31, odd d
;Out: a = algorithm
;     m = multiplier
;     s = shift factor

;algorithm 0
MOV   EDX, dividend
MOV   EAX, m
MUL   EDX
SHR   EDX, s   ;EDX=quotient

;algorithm 1
MOV   EDX, dividend
MOV   EAX, m
MUL   EDX
ADD   EAX, m
ADC   EDX, 0
SHR   EDX, s   ;EDX=quotient
```

Derivation of a, m, s

The derivation for the algorithm (a), multiplier (m), and shift count (s), is found in the section "Unsigned Derivation for Algorithm, Multiplier, and Shift Factor" on page 84.

Algorithm: Divisors $2^{31} <= d < 2^{32}$

For divisors $2^{31} <= d < 2^{32}$, there is no fancy code needed because the result is either 0 or 1. Therefore, for these divisors, the recommended code is as follows:

```
;In:  EDX = dividend
;Out: EDX = quotient
CMP    EDX, d      ;CF = (dividend < divisor) ? 1 : 0
SBB    EDX, EDX    ;(dividend < divisor) ? -1 : 0
INC    EDX         ;quotient = (dividend < divisor) ? 0 : 1
```

Simpler Code for Restricted Dividend

Integer division by a constant can be made faster if the range of the dividend is limited, which removes a shift associated with most divisors. For example, for a divide by 10 operation, use the following code if the dividend is less than 40000005h:

```
MOV   EAX, dividend
MOV   EDX, 01999999Ah
MUL   EDX
MOV   quotient, EDX
```

Signed Division by Multiplication of Constant

Algorithm: Divisors $2 <= d < 2^{31}$

These algorithms work if the divisor is positive. If the divisor is negative, use abs(d) instead of d, and append a 'NEG EDX' to the code. The code makes use of the fact that n/–d = –(n/d).

```
;IN:  d = divisor, 2 <= d < 2^31
;OUT: a = algorithm
;     m = multiplier
;     s = shift count

;algorithm 0
MOV   EAX, m
MOV   EDX, dividend
MOV   ECX, EDX
IMUL  EDX
SHR   ECX, 31
SAR   EDX, s
ADD   EDX, ECX           ;quotient in EDX

;algorithm 1
MOV   EAX, m
MOV   EDX, dividend
MOV   ECX, EDX
IMUL  EDX
ADD   EDX, ECX
SHR   ECX, 31
SAR   EDX, s
ADD   EDX, ECX           ;quotient in EDX
```

Derivation for a, m, s

The derivation for the algorithm (a), multiplier (m), and shift count (s), is found in the section "Signed Derivation for Algorithm, Multiplier, and Shift Factor" on page 86.

Signed Division By 2

```
;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

```
;IN:EAX = dividend
;OUT:EAX = quotient
CDQ                         ;Sign extend into EDX
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

```
;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 $-(2^n)$

```
;IN:EAX = dividend
;OUT:EAX = quotient
CDQ                         ;Sign extend into EDX
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

```
;IN:EAX = dividend
;OUT:EAX = remainder
CDQ                         ;Sign extend into EDX
AND   EDX, 1                ;Compute remainder
XOR   EAX, EDX              ;Negate remainder if
SUB   EAX, EDX              ;Dividend was < 0
MOV   [remainder], EAX
```

Remainder of Signed Integer 2^n or $-(2^n)$

```
;IN:EAX = dividend
;OUT:EAX = remainder
CDQ                       ;Sign extend into EDX
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   [remainder], EAX
```

Use Alternative Code When Multiplying by a Constant

A 32-bit integer multiply by a constant has a latency of five cycles. Therefore, use alternative code when multiplying by certain constants. In addition, because there is just one multiply unit, the replacement code may provide better throughput.

The following code samples are designed such that the original source also receives the final result. Other sequences are possible if the result is in a different register. Adds have been favored over shifts to keep code size small. Generally, there is a fast replacement if the constant has very few 1 bits in binary.

More constants are found in the file multiply_by_constants.txt located in the "opt_utilities" directory of the documentation CDROM.

```
by 2:    ADD   REG1, REG1              ;1 cycle

by 3:    LEA   REG1, [REG1*2+REG1]     ;2 cycles

by 4:    SHL   REG1, 2                 ;1 cycle

by 5:    LEA   REG1, [REG1*4+REG1]     ;2 cycles

by 6:    LEA   REG2, [REG1*4+REG1]     ;3 cycles
         ADD   REG1, REG2

by 7:    MOV   REG2, REG1              ;2 cycles
         SHL   REG1, 3
         SUB   REG1, REG2

by 8:    SHL   REG1, 3                 ;1 cycle

by 9:    LEA   REG1, [REG1*8+REG1]     ;2 cycles

by 10:   LEA   REG2, [REG1*8+REG1]     ;3 cycles
         ADD   REG1, REG2

by 11:   LEA   REG2, [REG1*8+REG1]     ;3 cycles
         ADD   REG1, REG1
         ADD   REG1, REG2

by 12:   SHL   REG1, 2
         LEA   REG1, [REG1*2+REG1]     ;3 cycles

by 13:   LEA   REG2, [REG1*2+REG1]     ;3 cycles
         SHL   REG1, 4
         SUB   REG1, REG2
```

```
by 14:   LEA   REG2, [REG1*4+REG1]     ;3 cycles
         LEA   REG1, [REG1*8+REG1]
         ADD   REG1, REG2

by 15:   MOV   REG2, REG1              ;2 cycles
         SHL   REG1, 4
         SUB   REG1, REG2

by 16:   SHL   REG1, 4                 ;1 cycle

by 17:   MOV   REG2, REG1              ;2 cycles
         SHL   REG1, 4
         ADD   REG1, REG2

by 18:   ADD   REG1, REG1              ;3 cycles
         LEA   REG1, [REG1*8+REG1]

by 19:   LEA   REG2, [REG1*2+REG1]     ;3 cycles
         SHL   REG1, 4
         ADD   REG1, REG2

by 20:   SHL   REG1, 2                 ;3 cycles
         LEA   REG1, [REG1*4+REG1]

by 21:   LEA   REG2, [REG1*4+REG1]     ;3 cycles
         SHL   REG1, 4
         ADD   REG1, REG2

by 22:   use IMUL

by 23:   LEA   REG2, [REG1*8+REG1]     ;3 cycles
         SHL   REG1, 5
         SUB   REG1, REG2

by 24:   SHL   REG1, 3                 ;3 cycles
         LEA   REG1, [REG1*2+REG1]

by 25:   LEA   REG2, [REG1*8+REG1]     ;3 cycles
         SHL   REG1, 4
         ADD   REG1, REG2

by 26:   use IMUL

by 27:   LEA   REG2, [REG1*4+REG1]     ;3 cycles
         SHL   REG1, 5
         SUB   REG1, REG2

by 28:   MOV   REG2, REG1              ;3 cycles
         SHL   REG1, 3
         SUB   REG1, REG2
         SHL   REG1, 2
```

```
by 29:   LEA   REG2, [REG1*2+REG1]     ;3 cycles
         SHL   REG1, 5
         SUB   REG1, REG2

by 30:   MOV   REG2, REG1              ;3 cycles
         SHL   REG1, 4
         SUB   REG1, REG2
         ADD   REG1, REG1

by 31:   MOV   REG2, REG1              ;2 cycles
         SHL   REG1, 5
         SUB   REG1, REG2

by 32:   SHL   REG1, 5                 ;1 cycle
```

Use MMX™ Instructions for Integer-Only Work

In many programs it can be advantageous to use MMX instructions to do integer-only work, especially if the function already uses 3DNow!™ or MMX code. Using MMX instructions relieves register pressure on the integer registers. As long as data is simply loaded/stored, added, shifted, etc., MMX instructions are good substitutes for integer instructions. Integer registers are freed up with the following results:

- May be able to reduce the number of integer registers to saved/restored on function entry/edit.
- Free up integer registers for pointers, loop counters, etc., so that they do not have to be spilled to memory, which reduces memory traffic and latency in dependency chains.

Be careful with regards to passing data between MMX and integer registers and of creating mismatched store-to-load forwarding cases. See “Unrolling Loops” on page 59.

In addition, using MMX instructions increases the available parallelism. The AMD Athlon processor can issue three integer OPs and two MMX OPs per cycle.

Repeated String Instruction Usage

Latency of Repeated String Instructions

Table 1 shows the latency for repeated string instructions on the AMD Athlon processor.

Table 1. Latency of Repeated String Instructions

Instruction	ECX=0 (cycles)	DF = 0 (cycles)	DF = 1 (cycles)
REP MOVS	11	15 + (4/3*c)	25 + (4/3*c)
REP STOS	11	14 + (1*c)	24 + (1*c)
REP LODS	11	15 + (2*c)	15 + (2*c)
REP SCAS	11	15 + (5/2*c)	15 + (5/2*c)
REP CMPS	11	16 + (10/3*c)	16 + (10/3*c)
Note: *c = value of ECX, (ECX > 0)*			

Table 1 lists the latencies with the direction flag (DF) = 0 (increment) and DF = 1. In addition, these latencies are assumed for aligned memory operands. Note that for MOVS/STOS, when DF = 1 (DOWN), the overhead portion of the latency increases significantly. However, these types are less commonly found. The user should use the formula and round up to the nearest integer value to determine the latency.

Guidelines for Repeated String Instructions

To help achieve good performance, this section contains guidelines for the careful scheduling of VectorPath repeated string instructions.

Use the Largest Possible Operand Size

Always move data using the largest operand size possible. For example, use REP MOVSD rather than REP MOVSW and REP MOVSW rather than REP MOVSB. Use REP STOSD rather than REP STOSW and REP STOSW rather than REP MOVSB.

Ensure DF=0 (UP)

Always make sure that DF = 0 (UP) (after execution of CLD) for REP MOVS and REP STOS. DF = 1 (DOWN) is only needed for certain cases of overlapping REP MOVS (for example, source and destination overlap).

While string instructions with DF = 1 (DOWN) are slower, only the overhead part of the cycle equation is larger and not the throughput part. See Table 1, "Latency of Repeated String Instructions," on page 76 for additional latency numbers.

Align Source and Destination with Operand Size

For REP MOVS, make sure that both source and destination are aligned with regard to the operand size. Handle the end case separately, if necessary. If either source or destination cannot be aligned, make the destination aligned and the source misaligned. For REP STOS, make the destination aligned.

Inline REP String with Low Counts

Expand REP string instructions into equivalent sequences of simple x86 instructions, if the repeat count is constant and less than eight. Use an inline sequence of loads and stores to accomplish the move. Use a sequence of stores to emulate REP STOS. This technique eliminates the setup overhead of REP instructions and increases instruction throughput.

Use Loop for REP String with Low Variable Counts

If the repeated count is variable, but is likely less than eight, use a simple loop to move/store the data. This technique avoids the overhead of REP MOVS and REP STOS.

Use MOVQs for Moving a Quadword Aligned Block of Data

For moving a large quadword-aligned block of data, use a partially unrolled loop that uses MOVQs. The following example is for Microsoft Visual C inline code and works equally well on the AMD-K6® and AMD Athlon processors.

Example:.

```
_asm {
mov   eax, [src]
mov   edx, [dst]
mov   ecx, (SIZE >> 6)

align 16

xfer:
movq  mm0, [eax]
add   edx, 64
movq  mm1, [eax+8]
add   eax, 64
movq  mm2, [eax-48]
movq  [edx-64], mm0
movq  mm3, [eax-40]
```

```
movq  [edx-56], mm1
movq  mm4, [eax-32]
movq  [edx-48], mm2
movq  mm5, [eax-24]
movq  [edx-40], mm3
movq  mm6, [eax-16]
movq  [edx-32], mm4
movq  mm7, [eax-8]
movq  [edx-24], mm5
movq  [edx-16], mm6
dec   ecx
movq  [edx-8], mm7
jnz   xfer
}
```

Use XOR Instruction to Clear Integer Registers

To clear an integer register to all 0s, use "XOR reg, reg". The AMD Athlon processor is able to avoid the false read dependency on the XOR instruction.

Example 1 (Acceptable):

```
MOV      REG, 0
```

Example 2 (Preferred):

```
XOR      REG, REG
```

Efficient 64-Bit Integer Arithmetic

This section contains a collection of code snippets and subroutines showing the efficient implementation of 64-bit arithmetic. Addition, subtraction, negation, and shifts are best handled by inline code. Multiplies, divides, and remainders are less common operations and should usually be implemented as subroutines. If these subroutines are used often, the programmer should consider inlining them. Except for division and remainder, the code presented works for both signed and unsigned integers. The division and remainder code shown works for unsigned integers, but can easily be extended to handle signed integers.

Example 1 (Addition):

```
;add operand in ECX:EBX to operand EDX:EAX, result in
;  EDX:EAX
ADD     EAX, EBX
ADC     EDX, ECX
```

Example 2 (Subtraction):

```
;subtract operand in ECX:EBX from operand EDX:EAX, result in
;  EDX:EAX
SUB     EAX, EBX
SBB     EDX, ECX
```

Example 3 (Negation):

```
;negate operand in EDX:EAX
NOT     EDX
NEG     EAX
SBB     EDX, -1 ;fixup: increment hi-word if low-word was 0
```

Example 4 (Left shift):

```
;shift operand in EDX:EAX left, shift count in ECX (count
;  applied modulo 64)
SHLD    EDX, EAX, CL      ;first apply shift count
SHL     EAX, CL           ;  mod 32 to EDX:EAX
TEST    ECX, 32           ;need to shift by another 32?
JZ      $lshift_done      ;no, done
MOV     EDX, EAX          ;left shift EDX:EAX
XOR     EAX, EAX          ;  by 32 bits

$lshift_done:
```

Example 5 (Right shift):

```
SHRD    EAX, EDX, CL      ;first apply shift count
SHR     EDX, CL           ;  mod 32 to EDX:EAX
TEST    ECX, 32           ;need to shift by another 32?
JZ      $rshift_done      ;no, done
MOV     EAX, EDX          ;left shift EDX:EAX
XOR     EDX, EDX          ;  by 32 bits

$rshift_done:
```

Example 6 (Multiplication):

```
;_llmul computes the low-order half of the product of its
;  arguments, two 64-bit integers
;
;INPUT:  [ESP+8]:[ESP+4]   multiplicand
;        [ESP+16]:[ESP+12] multiplier
;
;OUTPUT: EDX:EAX      (multiplicand * multiplier) % 2^64
;
;DESTROYS:  EAX,ECX,EDX,EFlags

_llmul PROC
MOV     EDX, [ESP+8]       ;multiplicand_hi
MOV     ECX, [ESP+16]      ;multiplier_hi
OR      EDX, ECX           ;one operand >= 2^32?
MOV     EDX, [ESP+12]      ;multiplier_lo
MOV     EAX, [ESP+4]       ;multiplicand_lo
JNZ     $twomul            ;yes, need two multiplies
MUL     EDX                ;multiplicand_lo * multiplier_lo
RET                        ;done, return to caller

$twomul:
IMUL    EDX, [ESP+8] ;p3_lo = multiplicand_hi*multiplier_lo
IMUL    ECX, EAX     ;p2_lo = multiplier_hi*multiplicand_lo
ADD     ECX, EDX     ; p2_lo + p3_lo
MUL     DWORD PTR [ESP+12] ;p1=multiplicand_lo*multiplier_lo
ADD     EDX, ECX     ;p1+p2lo+p3_lo = result in EDX:EAX
RET                  ;done, return to caller

_llmul ENDP
```

Example 7 (Division):

```
;_ulldiv divides two unsigned 64-bit integers, and returns
;  the quotient.
;
;INPUT:    [ESP+8]:[ESP+4]   dividend
;          [ESP+16]:[ESP+12] divisor
;
;OUTPUT:   EDX:EAX     quotient of division
;
;DESTROYS: EAX,ECX,EDX,EFlags
_ulldiv PROC
PUSH    EBX             ;save EBX as per calling convention
MOV     ECX, [ESP+20]   ;divisor_hi
MOV     EBX, [ESP+16]   ;divisor_lo
MOV     EDX, [ESP+12]   ;dividend_hi
MOV     EAX, [ESP+8]    ;dividend_lo
TEST    ECX, ECX        ;divisor > 2^32-1?
JNZ     $big_divisor    ;yes, divisor > 32^32-1
CMP     EDX, EBX        ;only one division needed? (ECX = 0)
JAE     $two_divs       ;need two divisions
DIV     EBX             ;EAX = quotient_lo
```

```
MOV     EDX, ECX        ;EDX = quotient_hi = 0 (quotient in
                        ;  EDX:EAX)
POP     EBX         ;restore EBX as per calling convention
RET                 ;done, return to caller

$two_divs:
MOV     ECX, EAX    ;save dividend_lo in ECX
MOV     EAX, EDX    ;get dividend_hi
XOR     EDX, EDX    ;zero extend it into EDX:EAX
DIV     EBX         ;quotient_hi in EAX
XCHG    EAX, ECX    ;ECX = quotient_hi, EAX = dividend_lo
DIV     EBX         ;EAX = quotient_lo
MOV     EDX, ECX    ;EDX = quotient_hi (quotient in EDX:EAX)
POP     EBX         ;restore EBX as per calling convention
RET                 ;done, return to caller

$big_divisor:
PUSH    EDI             ;save EDI as per calling convention
MOV     EDI, ECX        ;save divisor_hi
SHR     EDX, 1          ;shift both divisor and dividend right
RCR     EAX, 1          ; by 1 bit
ROR     EDI, 1
RCR     EBX, 1
BSR     ECX, ECX        ;ECX = number of remaining shifts
SHRD    EBX, EDI, CL    ;scale down divisor and dividend
SHRD    EAX, EDX, CL    ; such that divisor is less than
SHR     EDX, CL         ;   less than 2^32 (i.e. fits in EBX)
ROL     EDI, 1          ;restore original divisor_hi
DIV     EBX             ;compute quotient
MOV     EBX, [ESP+12]   ;dividend_lo
MOV     ECX, EAX        ;save quotient
IMUL    EDI, EAX        ;quotient * divisor hi-word
                        ;   (low only)
MUL     DWORD PTR [ESP+20];quotient * divisor lo-word
ADD     EDX, EDI            ;EDX:EAX = quotient * divisor
SUB     EBX, EAX            ;dividend_lo - (quot.*divisor)_lo
MOV     EAX, ECX            ;get quotient
MOV     ECX, [ESP+16]       ;dividend_hi
SBB     ECX, EDX    ;subtract divisor * quot. from dividend
SBB     EAX, 0      ;adjust quotient if remainder negative
XOR     EDX, EDX    ;clear hi-word of quot(EAX<=FFFFFFFFh)
POP     EDI         ;restore EDI as per calling convention
POP     EBX         ;restore EBX as per calling convention
RET                 ;done, return to caller

_ulldiv ENDP
```

Example 8 (Remainder):

```
;_ullrem divides two unsigned 64-bit integers, and returns
;  the remainder.
;
;INPUT:     [ESP+8]:[ESP+4]   dividend
;           [ESP+16]:[ESP+12] divisor
;
;OUTPUT:    EDX:EAX     remainder of division
;
;DESTROYS:  EAX,ECX,EDX,EFlags

_ullrem PROC
PUSH    EBX            ;save EBX as per calling convention
MOV     ECX, [ESP+20]  ;divisor_hi
MOV     EBX, [ESP+16]  ;divisor_lo
MOV     EDX, [ESP+12]  ;dividend_hi
MOV     EAX, [ESP+8]   ;dividend_lo
TEST    ECX, ECX       ;divisor > 2^32-1?
JNZ     $r_big_divisor  ;yes, divisor > 32^32-1
CMP     EDX, EBX       ;only one division needed? (ECX = 0)
JAE     $r_two_divs    ;need two divisions
DIV     EBX            ;EAX = quotient_lo
MOV     EAX, EDX       ;EAX = remainder_lo
MOV     EDX, ECX       ;EDX = remainder_hi = 0
POP     EBX            ;restore EBX as per calling convention
RET                    ;done, return to caller

$r_two_divs:
MOV     ECX, EAX    ;save dividend_lo in ECX
MOV     EAX, EDX    ;get dividend_hi
XOR     EDX, EDX    ;zero extend it into EDX:EAX
DIV     EBX         ;EAX = quotient_hi, EDX = intermediate
                    ;  remainder
MOV     EAX, ECX    ;EAX = dividend_lo
DIV     EBX         ;EAX = quotient_lo
MOV     EAX, EDX    ;EAX = remainder_lo
XOR     EDX, EDX    ;EDX = remainder_hi = 0
POP     EBX         ;restore EBX as per calling convention
RET                 ;done, return to caller

$r_big_divisor:
PUSH    EDI            ;save EDI as per calling convention
MOV     EDI, ECX       ;save divisor_hi
SHR     EDX, 1         ;shift both divisor and dividend right
RCR     EAX, 1         ; by 1 bit
ROR     EDI, 1
RCR     EBX, 1
BSR     ECX, ECX       ;ECX = number of remaining shifts
SHRD    EBX, EDI, CL   ;scale down divisor and dividend such
SHRD    EAX, EDX, CL   ; that divisor is less than 2^32
SHR     EDX, CL        ;  (i.e. fits in EBX)
```

```
ROL     EDI, 1          ;restore original divisor (EDI:ESI)
DIV     EBX             ;compute quotient
MOV     EBX, [ESP+12]   ;dividend lo-word
MOV     ECX, EAX        ;save quotient
IMUL    EDI, EAX        ;quotient * divisor hi-word (low only)
MUL     DWORD PTR [ESP+20] ;quotient * divisor lo-word
ADD     EDX, EDI        ;EDX:EAX = quotient * divisor
SUB     EBX, EAX        ;dividend_lo - (quot.*divisor)-lo
MOV     ECX, [ESP+16]   ;dividend_hi
MOV     EAX, [ESP+20]   ;divisor_lo
SBB     ECX, EDX        ;subtract divisor * quot. from
                        ;  dividend
SBB     EDX, EDX        ;(remainder < 0)? 0xFFFFFFFF : 0
AND     EAX, EDX        ;(remainder < 0)? divisor_lo : 0
AND     EDX, [ESP+24]   ;(remainder < 0)? divisor_hi : 0
ADD     EAX, EBX        ;remainder += (remainder < 0)?
ADC     EDX, ECX        ;   divisor : 0
POP     EDI             ;restore EDI as per calling convention
POP     EBX             ;restore EBX as per calling convention
RET                     ;done, return to caller

_ullrem ENDP
```

Derivation of Multiplier Used For Integer Division by Constants

Unsigned Derivation for Algorithm, Multiplier, and Shift Factor

The utility udiv.exe was compiled using the code shown in this section.

The following code derives the multiplier value used when performing integer division by constants. The code works for unsigned integer division and for odd divisors between 1 and $2^{31}-1$, inclusive. For divisors of the form d = d'*2^n, the multiplier is the same as for d' and the shift factor is s + n.

```
/* Code snippet to determine algorithm (a), multiplier (m),
and shift factor (s) to perform division on unsigned 32-bit
integers by constant divisor. Code is written for the
Microsoft Visual C compiler. */

/*
In:   d = divisor, 1 <= d < 2^31, d odd
Out:  a = algorithm
      m = multiplier
      s = shift factor

;algorithm 0
MOV   EDX, dividend
MOV   EAX, m
MUL   EDX
SHR   EDX, s   ;EDX=quotient

;algorithm 1
MOV   EDX, dividend
MOV   EAX, m
MUL   EDX
ADD   EAX, m
ADC   EDX, 0
SHR   EDX, s   ;EDX=quotient
*/

typedef unsigned __int64   U64;
typedef unsigned long      U32;

U32 d, l, s, m, a, r;
U64 m_low, m_high, j, k;
U32 log2 (U32 i)
{
```

```
  U32 t = 0;
  i = i >> 1;
  while (i) {
    i = i >> 1;
    t++;
  }
  return (t);
}

/* Generate m, s for algorithm 0. Based on: Granlund, T.;
Montgomery, P.L.:"Division by Invariant Integers using
Multiplication". SIGPLAN Notices, Vol. 29, June 1994, page
61. */

l       = log2(d) + 1;
j       = (((U64)(0xffffffff)) % ((U64)(d)));
k       = (((U64)(1)) << (32+l)) / ((U64)(0xffffffff-j));
m_low   = (((U64)(1)) << (32+l)) / d;
m_high  = ((((U64)(1)) << (32+l)) + k) / d;
while (((m_low >> 1) < (m_high >> 1)) && (l > 0)) {
   m_low  = m_low  >> 1;
   m_high = m_high >> 1;
   l      = l - 1;
}
if ((m_high >> 32) == 0) {
   m = ((U32)(m_high));
   s = l;
   a = 0;
}
/* Generate m, s for algorithm 1. Based on: Magenheimer,
D.J.; et al: "Integer Multiplication and Division on the HP
Precision Architecture". IEEE Transactions on Computers, Vol
37, No. 8, August 1988, page 980. */

else {
   s = log2(d);
   m_low = (((U64)(1)) << (32+s)) / ((U64)(d));
   r     = ((U32)((((U64)(1)) << (32+s)) % ((U64)(d))));
   m = (r < ((d>>1)+1)) ? ((U32)(m_low)) : ((U32)(m_low))+1;
   a = 1;
}

/* Reduce multiplier/shift factor for either algorithm to
smallest possible */

while (!(m&1)) {
   m = m >> 1;
   s--;
}
```

Signed Derivation for Algorithm, Multiplier, and Shift Factor

The utility sdiv.exe was compiled using the following code.

```
/* Code snippet to determine algorithm (a), multiplier (m),
and shift count (s) for 32-bit signed integer division,
given divisor d. Written for Microsoft Visual C compiler. */

/*
IN:   d = divisor, 2 <= d < 2^31
OUT:  a = algorithm
      m = multiplier
      s = shift count

;algorithm 0
MOV   EAX, m
MOV   EDX, dividend
MOV   ECX, EDX
IMUL  EDX
SHR   ECX, 31
SAR   EDX, s
ADD   EDX, ECX           ; quotient in EDX

;algorithm 1
MOV   EAX, m
MOV   EDX, dividend
MOV   ECX, EDX
IMUL  EDX
ADD   EDX, ECX
SHR   ECX, 31
SAR   EDX, s
ADD   EDX, ECX           ; quotient in EDX
*/

typedef unsigned __int64   U64;
typedef unsigned long      U32;

U32 log2 (U32 i)
{
  U32 t = 0;
  i = i >> 1;
  while (i) {
    i = i >> 1;
    t++;
  }
  return (t);
}

U32 d, l, s, m, a;
U64 m_low, m_high, j, k;
```

```
/* Determine algorithm (a), multiplier (m), and shift count
(s) for 32-bit signed integer division. Based on: Granlund,
T.; Montgomery, P.L.: “Division by Invariant Integers using
Multiplication”. SIGPLAN Notices, Vol. 29, June 1994, page
61. */

l       = log2(d);
j       = (((U64)(0x80000000)) % ((U64)(d)));
k       = (((U64)(1)) << (32+l)) / ((U64)(0x80000000-j));
m_low   = (((U64)(1)) << (32+l)) / d;
m_high  = ((((U64)(1)) << (32+l)) +  k) / d;
while (((m_low >> 1) < (m_high >> 1)) && (l > 0)) {
   m_low  = m_low  >> 1;
   m_high = m_high >> 1;
   l      = l - 1;
}
m = ((U32)(m_high));
s = l;
a = (m_high >> 31) ? 1 : 0;
```

9

Floating-Point Optimizations

This chapter details the methods used to optimize floating-point code to the pipelined floating-point unit (FPU). Guidelines are listed in order of importance.

Ensure All FPU Data is Aligned

As discussed in "Memory Size and Alignment Issues" on page 37, floating-point data should be naturally aligned. That is, words should be aligned on word boundaries, doublewords on doubleword boundaries, and quadwords on quadword boundaries. Misaligned memory accesses reduce the available memory bandwidth.

Use Multiplies Rather Than Divides

If accuracy requirements allow, floating-point division by a constant should be converted to a multiply by the reciprocal. Divisors that are powers of two and their reciprocal are exactly representable, except in the rare case that the reciprocal overflows or underflows, and therefore does not cause an accuracy issue. Unless such an overflow or underflow occurs, a division by a power of two should always be converted to a multiply. Although the AMD Athlon™ processor has high-performance division, multiplies are significantly faster than divides.

Use FFREEP Macro to Pop One Register from the FPU Stack

In FPU intensive code, frequently accessed data is often pre-loaded at the bottom of the FPU stack before processing floating-point data. After completion of processing, it is desirable to remove the pre-loaded data from the FPU stack as quickly as possible. The classical way to clean up the FPU stack is to use either of the following instructions:

```
FSTP     ST(0)       ;removes one register from stack

FCOMPP               ;removes two registers from stack
```

On the AMD Athlon processor, a faster alternative is to use the FFREEP instruction below. Note that the FFREEP instruction, although insufficiently documented in the past, is supported by all 32-bit x86 processors. The opcode bytes for FFREEP ST(i) are listed in Table 14 on page 166.

```
FFREEP   ST(0)       ;removes one register from stack
```

On the AMD Athlon processor, the FFREEP instruction converts to an internal NOP, which can go down any pipe with no dependencies.

Many assemblers do not support the FFREEP instruction. In these cases, a simple text macro can be created to facilitate use of the FFREEP ST(0).

```
FFREEP_ST0     TEXTEQU     <DB 0DFh, 0C0h>
```

Floating-Point Compare Instructions

For branches that are dependent on floating-point comparisons, use the FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions. These instructions are much faster than the classical approach using FSTSW, because FSTSW is essentially a serializing instruction on the AMD Athlon processor. When FSTSW cannot be avoided (for example, backward compatibility of code with older processors), no FPU instruction should occur between an FCOM[P], FICOM[P], FUCOM[P], or FTST and a dependent FSTSW. This optimization allows the use of a fast forwarding mechanism for the FPU condition codes internal to the AMD Athlon processor FPU and increases performance.

Use the FXCH Instruction Rather Than FST/FLD Pairs

Increase parallelism by breaking up dependency chains or by evaluating multiple dependency chains simultaneously by explicitly switching execution between them. Although the AMD Athlon processor FPU has a deep scheduler, which in most cases can extract sufficient parallelism from existing code, long dependency chains can stall the scheduler while issue slots are still available. The maximum dependency chain length that the scheduler can absorb is about six 4-cycle instructions.

To switch execution between dependency chains, use of the FXCH instruction is recommended because it has an apparent latency of zero cycles and generates only one OP. The AMD Athlon processor FPU contains special hardware to handle up to three FXCH instructions per cycle. Using FXCH is preferred over the use of FST/FLD pairs, even if the FST/FLD pair works on a register. An FST/FLD pair adds two cycles of latency and consists of two OPs.

Avoid Using Extended-Precision Data

Store data as either single-precision or double-precision quantities. Loading and storing extended-precision data is comparatively slower.

Minimize Floating-Point-to-Integer Conversions

C++, C, and Fortran define floating-point-to-integer conversions as truncating. This creates a problem because the active rounding mode in an application is typically round-to-nearest-even. The classical way to do a double-to-int conversion therefore works as follows:

Example 1 (Fast):

```
SUB    [I], EDX                ;trunc(X)=rndint(X)-correction
FLD    QWORD PTR [X]           ;load double to be converted
FSTCW  [SAVE_CW]               ;save current FPU control word
MOVZX  EAX, WORD PTR[SAVE_CW];retrieve control word
OR     EAX, 0C00h          ;rounding control field = truncate
```

```
MOV    WORD PTR [NEW_CW], AX ;new FPU control word
FLDCW  [NEW_CW]              ;load new FPU control word
FISTP  DWORD PTR [I]         ;do double->int conversion
FLDCW  [SAVE_CW]             ;restore original control word
```

The AMD Athlon processor contains special acceleration hardware to execute such code as quickly as possible. In most situations, the above code is therefore the fastest way to perform floating-point-to-integer conversion and the conversion is compliant both with programming language standards and the IEEE-754 standard.

The speed of the above code is somewhat dependent on the nature of the code surrounding it. For applications in which the speed of floating-point-to-integer conversions is extremely critical for application performance, experiment with either of the following substitutions, which may or may not be faster than the code above.

The first substitution simulates a truncating floating-point to integer conversion provided that there are no NaNs, infinities, and overflows. This conversion is therefore not IEEE-754 compliant. This code works properly only if the current FPU rounding mode is round-to-nearest-even, which is usually the case.

Example 2 (Potentially faster).

```
FLD    QWORD PTR [X]      ;load double to be converted
FST    DWORD PTR [TX]     ;store X because sign(X) is needed
FIST   DWORD PTR [I]      ;store rndint(x) as default result
FISUB  DWORD PTR [I]      ;compute DIFF = X - rndint(X)
FSTP   DWORD PTR [DIFF]   ;store DIFF as we need sign(DIFF)
MOV    EAX, [TX]          ;X
MOV    EDX, [DIFF]        ;DIFF
TEST   EDX, EDX           ;DIFF == 0 ?
JZ     $DONE              ;default result is OK, done
XOR    EDX, EAX   ; need correction if sign(X) != sign(DIFF)
SAR    EAX, 31            ;(X<0) ? 0xFFFFFFFF : 0
SAR    EDX, 31            ; sign(X)!=sign(DIFF)?0xFFFFFFFF:0
LEA    EAX, [EAX+EAX+1]   ;(X<0) ? 0xFFFFFFFF : 1
AND    EDX, EAX           ;correction: -1, 0, 1
SUB    [I], EDX           ;trunc(X)=rndint(X)-correction
$DONE:
```

The second substitution simulates a truncating floating-point to integer conversion using only integer instructions and therefore works correctly independent of the FPU's current rounding mode. It does not handle NaNs, infinities, and overflows

according to the IEEE-754 standard. Note that the first instruction of this code may cause an STLF size mismatch resulting in performance degradation if the variable to be converted has been stored recently.

Example 3 (Potentially faster):

```
MOV    ECX, DWORD PTR[X+4]   ;get upper 32 bits of double
XOR    EDX, EDX              ;i = 0
MOV    EAX, ECX              ;save sign bit
AND    ECX, 07FF00000h       ;isolate exponent field
CMP    ECX, 03FF00000h       ;if abs(x) < 1.0
JB     $DONE2                ; then i = 0
MOV    EDX, DWORD PTR[X]     ;get lower 32 bits of double
SHR    ECX, 20               ;extract exponent
SHRD   EDX, EAX, 21          ;extract mantissa
NEG    ECX             ;compute shift factor for extracting
ADD    ECX, 1054             ;non-fractional mantissa bits
OR     EDX, 080000000h       ;set integer bit of mantissa
SAR    EAX, 31               ;x < 0 ? 0xffffffff : 0
SHR    EDX, CL               ;i = trunc(abs(x))
XOR    EDX, EAX              ;i = x < 0 ? ~i : i
SUB    EDX, EAX              ;i = x < 0 ? -i : i
$DONE2:
MOV    [I], EDX              ;store result
```

For applications which can tolerate a floating-point-to-integer conversion that is not compliant with existing programming language standards (but is IEEE-754 compliant), perform the conversion using the rounding mode that is currently in effect (usually round-to-nearest-even).

Example 4 (Fastest):

```
FLD    QWORD PTR [X]      ; get double to be converted
FISTP  DWORD PTR [I]      ; store integer result
```

Some compilers offer an option to use the code from example 4 for floating-point-to-integer conversion, using the default rounding mode.

Lastly, consider setting the rounding mode throughout an application to truncate and using the code from example 4 to perform extremely fast conversions that are compliant with language standards and IEEE-754. This mode is also provided as an option by some compilers. Note that use of this technique also changes the rounding mode for all other FPU operations inside the application, which can lead to significant changes in numerical results and even program failure (for example, due to lack of convergence in iterative algorithms).

Floating-Point Subexpression Elimination

There are cases which do not require an FXCH instruction after every instruction to allow access to two new stack entries. In the cases where two instructions share a source operand, an FXCH is not required between the two instructions. When there is an opportunity for subexpression elimination, reduce the number of superfluous FXCH instructions by putting the shared source operand at the top of the stack. For example, using the function:

```
func( (x*y), (x+z) )
```

Example 1 (Avoid):

```
FLD     Z
FLD     Y
FLD     X
FADD    ST, ST(2)
FXCH    ST(1)
FMUL    ST, ST(2)
CALL    FUNC
FSTP    ST(0)
```

Example 2 (Preferred):

```
FLD     Z
FLD     Y
FLD     X
FMUL    ST(1), ST
FADDP   ST(2), ST
CALL    FUNC
```

10

3DNow!™ and MMX™ Optimizations

This chapter describes 3DNow! and MMX code optimization techniques for the AMD Athlon™ processor. Guidelines are listed in order of importance. 3DNow! porting guidelines can be found in the *3DNow!™ Instruction Porting Guide*, order# 22621.

Use 3DNow!™ Instructions

Unless accuracy requirements dictate otherwise, perform floating-point computations using the 3DNow! instructions instead of x87 instructions. The SIMD nature of 3DNow! achieves twice the number of FLOPs that are achieved through x87 instructions. 3DNow! instructions provide for a flat register file instead of the stack-based approach of x87 instructions.

See the *3DNow!™ Technology Manual*, order# 21928 for information on instruction usage.

Use FEMMS Instruction

Though there is no penalty for switching between x87 FPU and 3DNow!/MMX instructions in the AMD Athlon processor, the FEMMS instruction should be used to ensure the same code also runs optimally on the AMD-K6® processor. The FEMMS

instruction is supported for backward compatibility with the AMD-K6 processor, and is aliased to the EMMS instruction.

Use 3DNow!™ Instructions for Fast Division

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

Optimized 15-Bit Precision Divide

This divide operation executes with a total latency of seven cycles, assuming that the program hides the latency of the first MOVD/MOVQ instructions within preceding code.

Example:

```
MOVD     MM0, [MEM]  ;       0 | W
PFRCP    MM0, MM0    ;     1/W | 1/W
MOVQ     MM2, [MEM]  ;       Y | X
PFMUL    MM2, MM0    ;     Y/W | X/W
```

Optimized Full 24-Bit Precision Divide

This divide operation executes with a total latency of 15 cycles, assuming that the program hides the latency of the first MOVD/MOVQ instructions within preceding code.

Example:

```
MOVD       MM0, [W]        ;           0 | W
PFRCP      MM1, MM0        ;         1/W | 1/W
PUNPCKLDQ  MM0, MM0        ;           W | W
PFRCPIT1   MM0, MM1        ; 1/W refine | 1/W refine
MOVQ       MM2, [X_Y]      ;           Y | X
PFRCPIT2   MM0, MM1        ;  1/W final | 1/W final
PFMUL      MM2, MM0        ;         Y/W | X/W
```

Pipelined Pair of 24-Bit Precision Divides

This divide operation executes with a total latency of 21 cycles, assuming that the program hides the latency of the first MOVD/MOVQ instructions within preceding code.

Example:

```
MOVQ       MM0, [DIVISORS]   ;   y | x
PFRCP      MM1, MM0          ; 1/x | 1/x   (approximate)
MOVQ       MM2, MM0          ;   y | x
PUNPCKHDQ  MM0, MM0          ;   y | y
PFRCP      MM0, MM0          ; 1/y | 1/y   approximate
PUNPCKLDQ  MM1, MM0          ; 1/y | 1/x   approximate
MOVQ       MM0, [DIVIDENDS]  ;   z | w
PFRCPIT1   MM2, MM1          ; 1/y | 1/x   intermediate
PFRCPIT2   MM2, MM1          ; 1/y | 1/x   final
PFMUL      MM0, MM2          ; z/y | w/x
```

Newton-Raphson Reciprocal

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

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

$$X_{i+1} = X_i \bullet (2 - b \bullet X_i)$$

Given that the initial approximation, X_0, is accurate to at least 14 bits, and that a full IEEE single-precision mantissa contains 24 bits, just one Newton-Raphson iteration is required. The following sequence shows the 3DNow! instructions that produce the initial reciprocal approximation, compute the full precision reciprocal from the approximation, and finally, complete the desired divide of ${}^{a}/_{b}$.

X_0 = PFRCP(b)
X_1 = PFRCPIT1(b,X_0)
X_2 = PFRCPIT2(X_1,X_0)
q = PFMUL(a,X_2)

The 24-bit final reciprocal value is X_2. In the AMD Athlon processor 3DNow! technology implementation the operand X_2 contains the correct round-to-nearest single precision reciprocal for approximately 99% of all arguments.

Use 3DNow!™ Instructions for Fast Square Root and Reciprocal Square Root

3DNow! instructions can be used to compute a very fast, highly accurate square root and reciprocal square root.

Optimized 15-Bit Precision Square Root

This square root operation can be executed in only 7 cycles, assuming a program hides the latency of the first MOVD instruction within previous code. The reciprocal square root operation requires four less cycles than the square root operation.

Example:

```
MOVD      MM0, [MEM]    ;        0 | a
PFRSQRT   MM1, MM0      ;1/sqrt(a) | 1/sqrt(a) (approximate)
PUNPCKLDQ MM0, MM0      ;        a | a         (MMX instr.)
PFMUL     MM0, MM1      ;  sqrt(a) | sqrt(a)
```

Optimized 24-Bit Precision Square Root

This square root operation can be executed in only 19 cycles, assuming a program hides the latency of the first MOVD instruction within previous code. The reciprocal square root operation requires four less cycles than the square root operation.

Example:

```
MOVD      MM0, [MEM]    ;        0 | a
PFRSQRT   MM1, MM0      ;1/sqrt(a) | 1/sqrt(a) (approximate)
MOVQ      MM2, MM1      ; X_0 = 1/(sqrt a)     (approximate)
PFMUL     MM1, MM1      ; X_0 * X_0 | X_0 * X_0  (step 1)
PUNPCKLDQ MM0, MM0      ;        a | a         (MMX instr.)
PFRSQIT1  MM1, MM0      ; intermediate         (step 2)
PFRCPIT2  MM1, MM2      ;1/sqrt(a) | 1/sqrt(a)(step 3)
PFMUL     MM0, MM1      ;  sqrt(a) | sqrt(a)
```

Newton-Raphson Reciprocal Square Root

The general Newton-Raphson reciprocal square root recurrence is:

$$X_{i+1} = 1/2 \bullet X_i \bullet (3 - b \bullet X_i^2)$$

To reduce the number of iterations, X_0 is an initial approximation read from a table. The 3DNow! reciprocal square root approximation is accurate to at least 15 bits. Accordingly, to obtain a single-precision 24-bit reciprocal square root of an input operand b, one Newton-Raphson iteration is required, using the following sequence of 3DNow! instructions:

X_0 = PFRSQRT(b)
X_1 = PFMUL(X_0,X_0)
X_2 = PFRSQIT1(b,X_1)
X_3 = PFRCPIT2(X_2,X_0)
X_4 = PFMUL(b,X_3)

The 24-bit final reciprocal square root value is X_3. In the AMD Athlon processor 3DNow! implementation, the estimate contains the correct round-to-nearest value for approximately 87% of all arguments. The remaining arguments differ from the correct round-to-nearest value by one unit-in-the-last-place. The square root (X_4) is formed in the last step by multiplying by the input operand b.

Use MMX™ PMADDWD Instruction to Perform Two 32-Bit Multiplies in Parallel

The MMX PMADDWD instruction can be used to perform two signed 16x16→32 bit multiplies in parallel, with much higher performance than can be achieved using the IMUL instruction. The PMADDWD instruction is designed to perform four 16x16→32 bit signed multiplies and accumulate the results pairwise. By making one of the results in a pair a zero, there are now just two multiplies. The following example shows how to multiply 16-bit signed numbers a,b,c,d into signed 32-bit products a×c and b×d:

Example:

```
PXOR      MM2, MM2       ;   0 | 0
MOVD      MM0, [ab]      ; 0 0 | b a
MOVD      MM1, [cd]      ; 0 0 | d c
PUNPCKLWD MM0, MM2       ; 0 b | 0 a
PUNCPKLWD MM1, MM2       ; 0 d | 0 c
PMADDWD   MM0, MM1       ; b*d | a*c
```

3DNow!™ and MMX™ Intra-Operand Swapping

If the swapping of MMX register halves is necessary, use the PSWAPD instruction, which is a new AMD Athlon MMX extension. Use of this instruction should only be for AMD Athlon specific code. See the *AMD Extensions to the 3DNow! and MMX Instruction Set Manual*, order #22466 for correct usage of this instruction.

Otherwise, for blended code, which needs to run well on AMD-K6 and AMD Athlon family processors, the following code is recommended:

Example 1 (Preferred, faster):

```
;MM1 = SWAP (MM0), MM0 destroyed
MOVQ        MM1, MM0       ;make a copy
PUNPCKLDQ   MM0, MM0       ;duplicate lower half
PUNPCKHDQ   MM1, MM0       ;combine lower halves
```

Example 2 (Preferred, fast):

```
;MM1 = SWAP (MM0), MM0 preserved
MOVQ        MM1, MM0       ;make a copy
PUNPCKHDQ   MM1, MM1       ;duplicate upper half
PUNPCKLDQ   MM1, MM0       ;combine upper halves
```

Both examples accomplish the swapping, but the first example should be used if the original contents of the register do not need to be preserved. The first example is faster due to the fact that the MOVQ and PUNPCKLDQ instructions can execute in parallel. The instructions in the second example are dependent on one another and take longer to execute.

Fast Conversion of Signed Words to Floating-Point

In many applications there is a need to quickly convert data consisting of packed 16-bit signed integers into floating-point

numbers. The following two examples show how this can be accomplished efficiently on AMD processors.

The first example shows how to do the conversion on a processor that supports AMD's 3DNow! extensions, such as the AMD Athlon processor. It demonstrates the increased efficiency from using the PI2FW instruction. Use of this instruction should only be for AMD Athlon specific code. See the *AMD Extensions to the 3DNow! and MMX Instruction Set Manual*, order #22466 for more information on this instruction.

The second example demonstrates how to accomplish the same task in blended code that achieves good performance on the AMD Athlon processor as well as on the AMD-K6 family processors that support 3DNow! technology.

Example 1 (AMD Athlon specific code using 3DNow! DSP instruction):

```
MOVD      MM0, [packed_sword]    ;0 0 | b a
PUNPCKLWD MM0, MM0               ;b b | a a
PI2FW     MM0, MM0               ;xb=float(b)|xa=float(a)
MOVQ      [packed_float], MM0    ;store xb | xa
```

Example 2 (AMD K6 Family and AMD Athlon blended code):

```
MOVD      MM1, [packed_sword]   ;0 0 | b a
PXOR      MM0, MM0              ;0 0 | 0 0
PUNPCKLWD MM0, MM1              ;b 0 | a 0
PSRAD     MM0, 16               ;sign extend: b | a
PI2FD     MM0, MM0              ;xb=float(b) | xa=float(a)
MOVQ      [packed_float], MM0   ;store xb | xa
```

Use MMX™ PXOR to Change the Sign Bit in 3DNow!™ Code

For both the AMD Athlon and AMD-K6 processors, it is recommended that code use the MMX PXOR instruction to change the sign bit of 3DNow! operations instead of the 3DNow! PFMUL instruction. On the AMD Athlon processor, using PXOR allows for more parallelism, as it can execute in either the FADD or FMUL pipes. PXOR has an execution latency of two, but because it is a MMX instruction, there is an initial one cycle bypassing penalty, and another one cycle penalty if the result goes to a 3DNow! operation. The PFMUL execution latency is four, therefore, in the worst case, the PXOR and PMUL instructions are the same in terms of latency. On the AMD-K6 processor, there is only a one cycle latency for PXOR, versus a two cycle latency for the 3DNow! PFMUL instruction.

Use MMX™ PCMP Instead of 3DNow!™ PFCMP

Use the MMX PCMP instruction instead of the 3DNow! PFCMP instruction. On the AMD Athlon processor, the PCMP has a latency of two cycles while the PFCMP has a latency of four cycles. In addition to the shorter latency, PCMP can be issued to either the FADD or the FMUL pipe, while PFCMP is restricted to the FADD pipe.

Note: *The PFCMP instruction has a 'GE' (greater or equal) version (PFCMPGE) that is missing from PCMP.*

Both Numbers Positive

If both arguments are positive, PCMP always works.

One Negative, One Positive

If one number is negative and the other is positive, PCMP still works, except when one number is a positive zero and the other is a negative zero.

Both Numbers Negative

Be careful when performing integer comparison using PCMPGT on two negative 3DNow! numbers. The result is the inverse of the PFCMPGT floating-point comparison. For example:

```
-2 = 84000000
-4 = 84800000
```

PCMP gives 84800000 > 84000000, but –4 < –2. To address this issue, simply reverse the comparison by swapping the source operands.

Use MMX™ PXOR to Clear an MMX Register

To set all the bits in an MMX register to 0s, use:

```
PXOR mmreg, mmreg
```

Note that "PXOR mmreg, mmreg" is dependent on previous writes to mmreg. Therefore, using PXOR in the manner described can lengthen dependency chains, which in return may lead to reduced performance. In such instances, a MOVD should be used to load a zero from a statically initialized memory location.

Use MMX™ PCMPEQD to Set an MMX Register

To set all the bits in an MMX register to 1s, use:

```
PCMPEQD mmreg, mmreg
```

Note that "PCMPEQD mmreg, mmreg" is dependent on previous writes to mmreg. Therefore, using PCMPEQD in the manner described can lengthen dependency chains, which in return may lead to reduce performance. In such instances, MOVQ should be used to load the constant 0xFFFFFFFFFFFFFFFF from a statically initialized memory location.

Use MMX™ PAND to Find Absolute Value in 3DNow!™ Code

Use the following to compute the absolute value of 3DNow! floating-point operands:

```
mabs    DQ 7FFFFFFF7FFFFFFFh
PAND    MM0, [mabs]             ;mask out sign bit
```

Use MMX™ PXOR to Negate 3DNow!™ Data

Use the following code to negate 3DNow! data:

```
msgn    DQ 8000000080000000h
PXOR    MM0, [msgn]             ;toggle sign bit
```

Use 3DNow!™ PAVGUSB for MPEG-2 Motion Compensation

Use the 3DNow! PAVGUSB instruction for MPEG-2 motion compensation. The PAVGUSB instruction produces the rounded averages of the eight unsigned 8-bit integer values in the source operand (a MMX register or a 64-bit memory location) and the eight corresponding unsigned 8-bit integer values in the destination operand (a MMX register). The PAVGUSB instruction is extremely useful in DVD (MPEG-2) decoding where motion compensation performs a lot of byte averaging between and within macroblocks. The PAVGUSB instruction

helps speed up these operations. In addition, PAVGUSB can free up some registers and make unrolling the averaging loops possible.

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

Example 1 (Avoid):

```
MOV      ESI, DWORD PTR Src_MB
MOV      EDI, DWORD PTR Dst_MB
MOV      EDX, DWORD PTR SrcStride
MOV      EBX, DWORD PTR DstStride
MOVQ     MM7, QWORD PTR [ConstFEFE]
MOVQ     MM6, QWORD PTR [Const0101]
MOV      ECX, 16

L1:
MOVQ     MM0, [ESI]          ;MM0=QWORD1
MOVQ     MM1, [EDI]          ;MM1=QWORD3
MOVQ     MM2, MM0
MOVQ     MM3, MM1
PAND     MM2, MM6
PAND     MM3, MM6
PAND     MM0, MM7            ;MM0 = QWORD1 & 0xfefefefe
PAND     MM1, MM7            ;MM1 = QWORD3 & 0xfefefefe
POR      MM2, MM3            ;calculate adjustment
PSRLQ    MM0, 1              ;MM0 = (QWORD1 & 0xfefefefe)/2
PSRLQ    MM1, 1              ;MM1 = (QWORD3 & 0xfefefefe)/2
PAND     MM2, MM6
PADDB    MM0, MM1            ;MM0 = QWORD1/2 + QWORD3/2 w/o
                             ; adjustment
PADDB    MM0, MM2            ;add lsb adjustment
MOVQ     [EDI], MM0
MOVQ     MM4, [ESI+8]        ;MM4=QWORD2
MOVQ     MM5, [EDI+8]        ;MM5=QWORD4
MOVQ     MM2, MM4
MOVQ     MM3, MM5
PAND     MM2, MM6
PAND     MM3, MM6
PAND     MM4, MM7            ;MM0 = QWORD2 & 0xfefefefe
PAND     MM5, MM7            ;MM1 = QWORD4 & 0xfefefefe
POR      MM2, MM3            ;calculate adjustment
PSRLQ    MM4, 1              ;MM0 = (QWORD2 & 0xfefefefe)/2
PSRLQ    MM5, 1              ;MM1 = (QWORD4 & 0xfefefefe)/2
PAND     MM2, MM6
PADDB    MM4, MM5            ;MM0 = QWORD2/2 + QWORD4/2 w/o
                             ; adjustment
PADDB    MM4, MM2            ;add lsb adjustment
MOVQ     [EDI+8], MM4
```

```
ADD       ESI, EDX
ADD       EDI, EBX
LOOP      L1
```

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

Example 2 (Preferred):

```
MOV       EAX, DWORD PTR Src_MB
MOV       EDI, DWORD PTR Dst_MB
MOV       EDX, DWORD PTR SrcStride
MOV       EBX, DWORD PTR DstStride
MOV       ECX, 16

L1:
MOVQ      MM0, [EAX]        ;MM0=QWORD1
MOVQ      MM1, [EAX+8]      ;MM1=QWORD2
PAVGUSB   MM0, [EDI]        ;(QWORD1 + QWORD3)/2 with
                            ; adjustment
PAVGUSB   MM1, [EDI+8]      ;(QWORD2 + QWORD4)/2 with
                            ; adjustment
ADD       EAX, EDX
MOVQ      [EDI], MM0
MOVQ      [EDI+8], MM1
ADD       EDI, EBX
LOOP      L1
```

Stream of Packed Unsigned Bytes

The following code is an example of how to process a stream of packed unsigned bytes (like RGBA information) with faster 3DNow! instructions.

Example:

```
outside loop:
PXOR        MM0, MM0

inside loop:
MOVD        MM1, [VAR]  ;          0 | v[3],v[2],v[1],v[0]
PUNPCKLBW   MM1, MM0    ;0,v[3],0,v[2] | 0,v[1],0,v[0]
MOVQ        MM2, MM1    ;0,v[3],0,v[2] | 0,v[1],0,v[0]
PUNPCKLWD   MM1, MM0    ;   0,0,0,v[1] | 0,0,0,v[0]
PUNPCKHWD   MM2, MM0    ;   0,0,0,v[3] | 0,0,0,v[2]
PI2FD       MM1, MM1    ;  float(v[1]) | float(v[0])
PI2FD       MM2, MM2    ;  float(v[3]) | float(v[2])
```

11

General x86 Optimization Guidelines

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

Example 1 (Avoid):

```
CMP      REG, 0
```

Example 2 (Preferred):

```
TEST     REG, REG
```

Although both of these instructions have an execute latency of one, fewer opcode bytes need to be examined by the decoders for the TEST instruction.

Dependencies

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

Register Operands

Maintain frequently used values in registers rather than in memory. This technique avoids the comparatively long latencies for accessing memory.

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. In addition, this method reduces ESP dependencies and uses fewer execution resources.

Appendix A

AMD Athlon™ Processor Microarchitecture

Introduction

When discussing processor design, it is important to understand the following 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 Athlon 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 Athlon processor microarchitecture is a decoupled decode/execution design approach. In other words, the decoders essentially operate independent of the execution units, and the execution core uses a small number of instructions and simplified circuit design for fast single-cycle execution and fast operating frequencies.

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

AMD Athlon™ Processor Microarchitecture

The innovative AMD Athlon processor microarchitecture approach implements the x86 instruction set by processing simpler operations (OPs) instead of complex x86 instructions. These OPs are specially designed to include direct support for the x86 instructions while observing the high-performance principles of fixed-length encoding, regularized instruction fields, and a large register set. Instead of executing complex x86 instructions, which have lengths from 1 to 15 bytes, the AMD Athlon processor executes the simpler fixed-length OPs, while maintaining the instruction coding efficiencies found in x86 programs. The enhanced microarchitecture used in the AMD Athlon processor enables higher processor core performance and promotes straightforward extendibility for future designs.

Superscalar Processor

The AMD Athlon processor is an aggressive, out-of-order, three-way superscalar x86 processor. It can fetch, decode, and issue up to three x86 instructions per cycle with a centralized instruction control unit (ICU) and two independent instruction schedulers—an integer scheduler and a floating-point scheduler. These two schedulers can simultaneously issue up to nine OPs to the three general-purpose integer execution units (IEUs), three address-generation units (AGUs), and three floating-point/3DNow!™/MMX™ execution units. The AMD Athlon moves integer instructions down the integer execution pipeline, which consists of the integer scheduler and the IEUs, as shown in Figure 1 on page 111. Floating-point instructions are handled by the floating-point execution pipeline, which consists of the floating-point scheduler and the x87/3DNow!/MMX execution units.

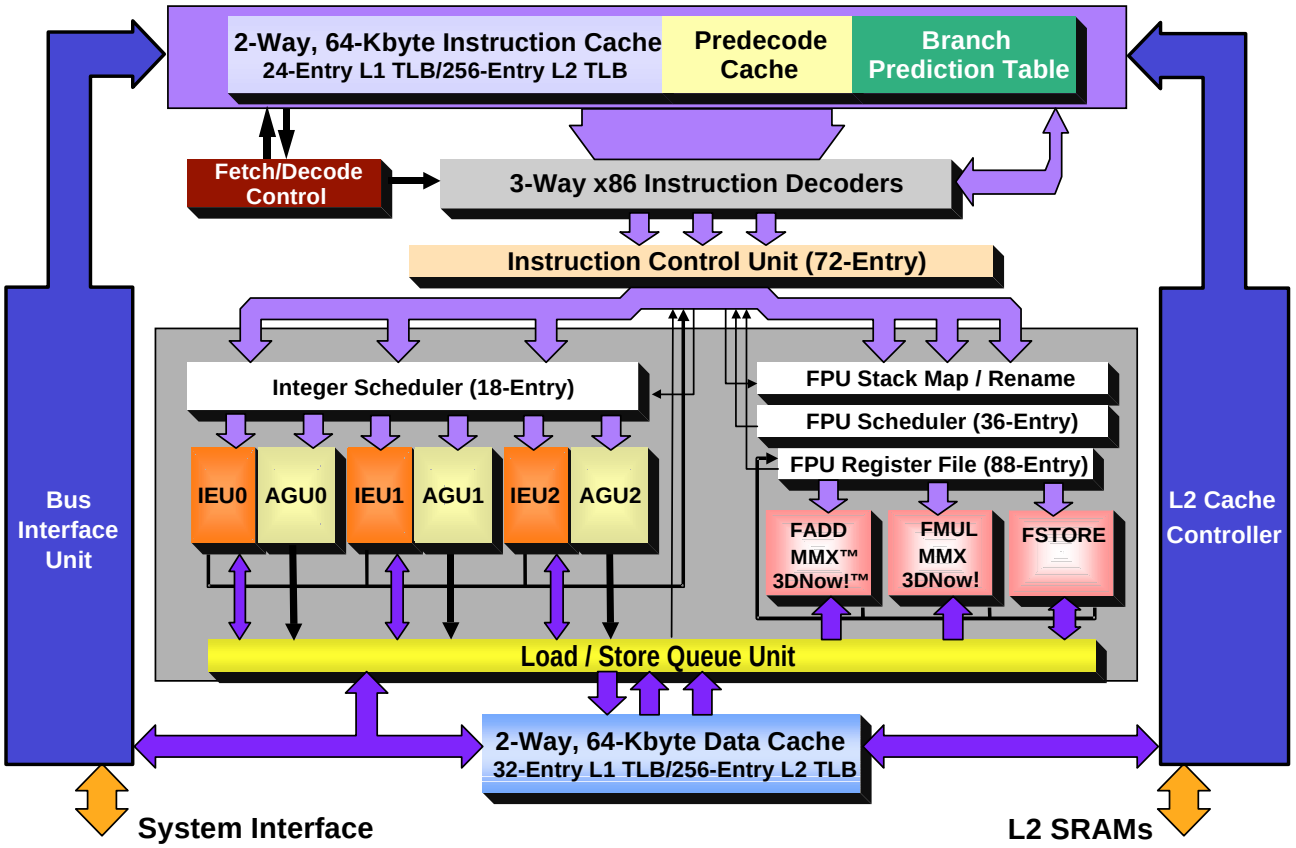

Figure 1. AMD Athlon™ Processor Block Diagram

Instruction Cache

The out-of-order execute engine of the AMD Athlon processor contains a very large 64-Kbyte L1 instruction cache. The L1 instruction cache is organized as a 64-Kbyte, two-way, set-associative array. Each line in the instruction array is 64 bytes long. Functions associated with the L1 instruction cache are instruction loads, instruction prefetching, instruction predecoding, and branch prediction. Requests that miss in the L1 instruction cache are fetched from the backside L2 cache or, subsequently, from the local memory using the bus interface unit (BIU).

The instruction cache generates fetches on the naturally aligned 64 bytes containing the instructions and the next sequential line of 64 bytes (a prefetch). The principal of *program spatial locality* makes data prefetching very effective and avoids or reduces execution stalls due to the amount of time wasted reading the necessary data. Cache line

replacement is based on a least-recently used (LRU) replacement algorithm.

The L1 instruction cache has an associated two-level translation look-aside buffer (TLB) structure. The first-level TLB is fully associative and contains 24 entries (16 that map 4-Kbyte pages and eight that map 2-Mbyte or 4-Mbyte pages). The second-level TLB is four-way set associative and contains 256 entries, which can map 4-Kbyte pages.

Predecode

Predecoding begins as the L1 instruction cache is filled. Predecode information is generated and stored alongside the instruction cache. This information is used to help efficiently identify the boundaries between variable length x86 instructions, to distinguish DirectPath from VectorPath early-decode instructions, and to locate the opcode byte in each instruction. In addition, the predecode logic detects code branches such as CALLs, RETURNs and short unconditional JMPs. When a branch is detected, predecoding begins at the target of the branch.

Branch Prediction

The fetch logic accesses the branch prediction table in parallel with the instruction cache and uses the information stored in the branch prediction table to predict the direction of branch instructions.

The AMD Athlon processor employs combinations of a branch target address buffer (BTB), a global history bimodal counter (GHBC) table, and a return address stack (RAS) hardware in order to predict and accelerate branches. Predicted-taken branches incur only a single-cycle delay to redirect the instruction fetcher to the target instruction. In the event of a mispredict, the minimum penalty is ten cycles.

The BTB is a 2048-entry table that caches in each entry the predicted target address of a branch.

In addition, the AMD Athlon processor implements a 12-entry return address stack to predict return addresses from a near or far call. As CALLs are fetched, the next EIP is pushed onto the

return stack. Subsequent RETs pop a predicted return address off the top of the stack.

Early Decoding

The DirectPath and VectorPath decoders perform early-decoding of instructions into MacroOPs. A MacroOP is a fixed length instruction which contains one or more OPs. The outputs of the early decoders keep all (DirectPath or VectorPath) instructions in program order. Early decoding produces three MacroOPs per cycle from either path. The outputs of both decoders are multiplexed together and passed to the next stage in the pipeline, the instruction control unit.

When the target 16-byte instruction window is obtained from the instruction cache, the predecode data is examined to determine which type of basic decode should occur—DirectPath or VectorPath.

DirectPath Decoder

DirectPath instructions can be decoded directly into a MacroOP, and subsequently into one or two OPs in the final issue stage. A DirectPath instruction is limited to those x86 instructions that can be further decoded into one or two OPs. The length of the x86 instruction does *not* determine DirectPath instructions. A maximum of three DirectPath x86 instructions can occupy a given aligned 8-byte block. 16-bytes are fetched at a time. Therefore, up to six DirectPath x86 instructions can be passed into the DirectPath decode pipeline.

VectorPath Decoder

Uncommon x86 instructions requiring two or more MacroOPs proceed down the VectorPath pipeline. The sequence of MacroOPs is produced by an on-chip ROM known as the MROM. The VectorPath decoder can produce up to three MacroOPs per cycle. Decoding a VectorPath instruction may prevent the simultaneous decode of a DirectPath instruction.

Instruction Control Unit

The instruction control unit (ICU) is the control center for the AMD Athlon processor. The ICU controls the following resources—the centralized in-flight reorder buffer, the integer scheduler, and the floating-point scheduler. In turn, the ICU is responsible for the following functions—MacroOP dispatch,

MacroOP retirement, register and flag dependency resolution and renaming, execution resource management, interrupts, exceptions, and branch mispredictions.

The ICU takes the three MacroOPs per cycle from the early decoders and places them in a centralized, fixed-issue reorder buffer. This buffer is organized into 24 lines of three MacroOPs each. The reorder buffer allows the ICU to track and monitor up to 72 in-flight MacroOPs (whether integer or floating-point) for maximum instruction throughput. The ICU can simultaneously dispatch multiple MacroOPs from the reorder buffer to both the integer and floating-point schedulers for final decode, issue, and execution as OPs. In addition, the ICU handles exceptions and manages the retirement of MacroOPs.

Data Cache

The L1 data cache contains two 64-bit ports. It is a write-allocate and writeback cache that uses an LRU replacement policy. The data cache and instruction cache are both two-way set-associative and 64-Kbytes in size. In addition, this cache supports the MOESI (Modified, Owner, Exclusive, Shared, and Invalid) cache coherency protocol and data parity.

The L1 data cache has an associated two-level TLB structure. The first-level TLB is fully associative and contains 32 entries (24 that map 4-Kbyte pages and eight that map 2-Mbyte or 4-Mbyte pages). The second-level TLB is four-way set associative and contains 256 entries, which can map 4-Kbyte pages.

Integer Scheduler

The integer scheduler is based on a three-wide queuing system (also known as a reservation station) that feeds three integer execution positions or pipes. Each reservation station divides the MacroOPs into integer and address generation OPs, as required.

Integer Execution Unit

The integer execution pipeline consists of three identical pipes—0, 1, and 2. Each integer pipe consists of an integer execution unit (IEU) and an address generation unit (AGU). The integer execution pipeline is organized to match the three MacroOP dispatch pipes in the ICU as shown in Figure 2 on page 115. MacroOPs are broken down into OPs in the schedulers. OPs issue when their operands are available either from the register file or result buses.

OPs are executed when their operands are available. OPs from a single MacroOP can execute out-of-order. In addition, a particular integer pipe can be executing two OPs from different MacroOPs (one in the IEU and one in the AGU) at the same time.

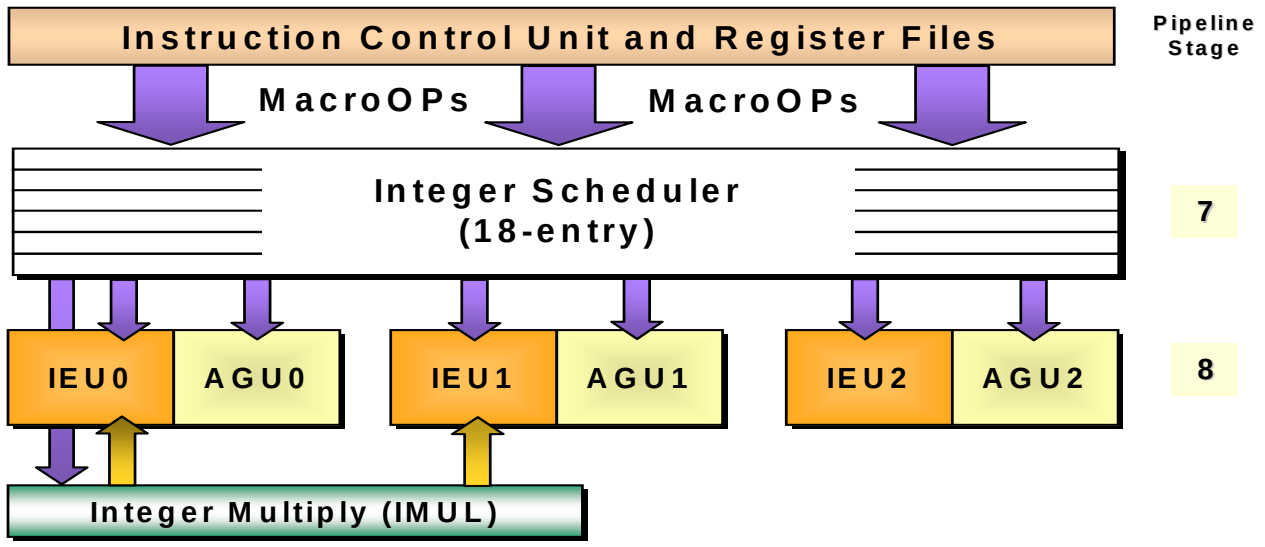

Figure 2. Integer Execution Pipeline

Each of the three IEUs are general purpose in that each performs logic functions, arithmetic functions, conditional functions, divide step functions, status flag multiplexing, and branch resolutions. The AGUs calculate the logical addresses for loads, stores, and LEAs. A load and store unit reads and writes data to and from the L1 data cache. The integer scheduler sends a completion status to the ICU when the outstanding OPs for a given MacroOP are executed.

All integer operations can be handled within any of the three IEUs with the exception of multiplies. Multiplies are handled by a pipelined multiplier that is attached to the pipeline at pipe 0. See Figure 2 on page 115.

Floating-Point Scheduler

The AMD Athlon processor floating-point logic is a high-performance, fully-pipelined, superscalar, out-of-order execution unit. It is capable of accepting three MacroOPs of any mixture of x87 floating-point, 3DNow! or MMX operations per cycle.

The floating-point scheduler handles register renaming and has a dedicated 36-entry scheduler buffer organized as 12 lines of three MacroOPs each. It also performs OP issue, and out-of-order execution. The floating-point scheduler communicates with the ICU to retire a MacroOP, to manage comparison results from the FCOMI instruction, and to back out results from a branch misprediction.

Floating-Point Execution Unit

The floating-point execution unit (FPU) is implemented as a coprocessor that has its own out-of-order control in addition to the data path. The FPU handles all register operations for x87 instructions, all 3DNow! operations, and all MMX operations. The FPU consists of a stack renaming unit, a register renaming unit, a scheduler, a register file, and three parallel execution units. Figure 3 shows a block diagram of the dataflow through the FPU.

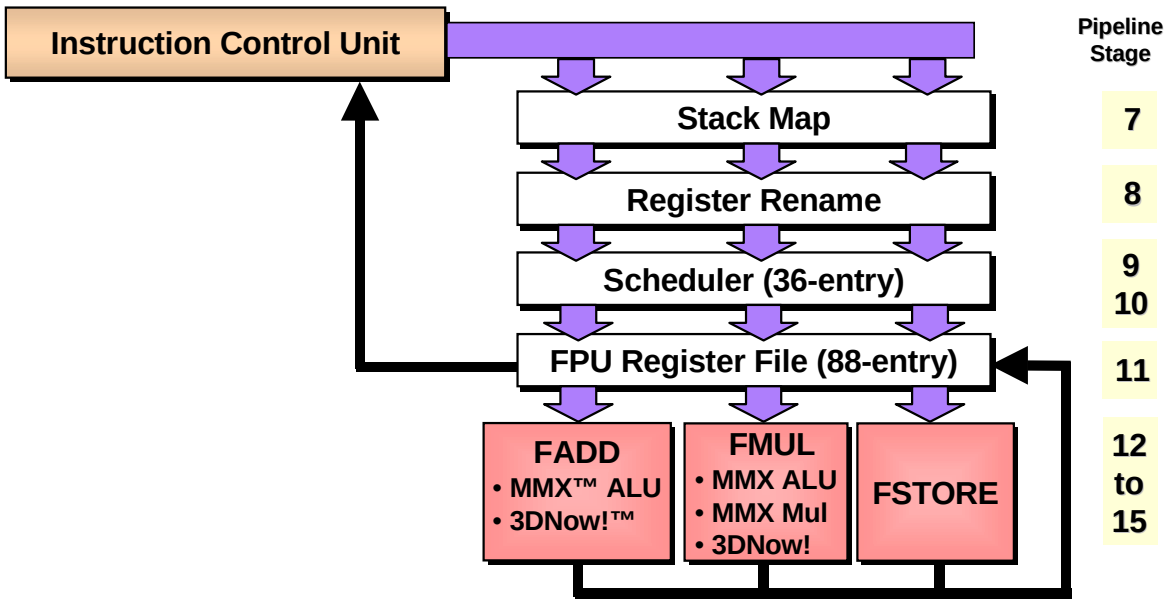

Figure 3. Floating-Point Unit Block Diagram

As shown in Figure 3 on page 117, the floating-point logic uses three separate execution positions or pipes for superscalar x87, 3DNow! and MMX operations. The first of the three pipes is generally known as the adder pipe (FADD), and it contains 3DNow! add, MMX ALU/shifter, and floating-point add execution units. The second pipe is known as the multiplier (FMUL). It contains a 3DNow!/MMX multiplier/reciprocal unit, an MMX ALU and a floating-point multiplier/divider/square root unit. The third pipe is known as the floating-point load/store (FSTORE), which handles floating-point constant loads (FLDZ, FLDPI, etc.), stores, FILDs, as well as many OP primitives used in VectorPath sequences.

Load-Store Unit (LSU)

The load-store unit (LSU) manages data load and store accesses to the L1 data cache and, if required, to the backside L2 cache or system memory. The 44-entry LSU provides a data interface for both the integer scheduler and the floating-point scheduler. It consists of two queues—a 12-entry queue for L1 cache load and store accesses and a 32-entry queue for L2 cache or system memory load and store accesses. The 12-entry queue can request a maximum of two L1 cache loads and two L1 cache (32-bits) stores per cycle. The 32-entry queue effectively holds

requests that missed in the L1 cache probe by the 12-entry queue. Finally, the LSU ensures that the architectural load and store ordering rules are preserved (a requirement for x86 architecture compatibility).

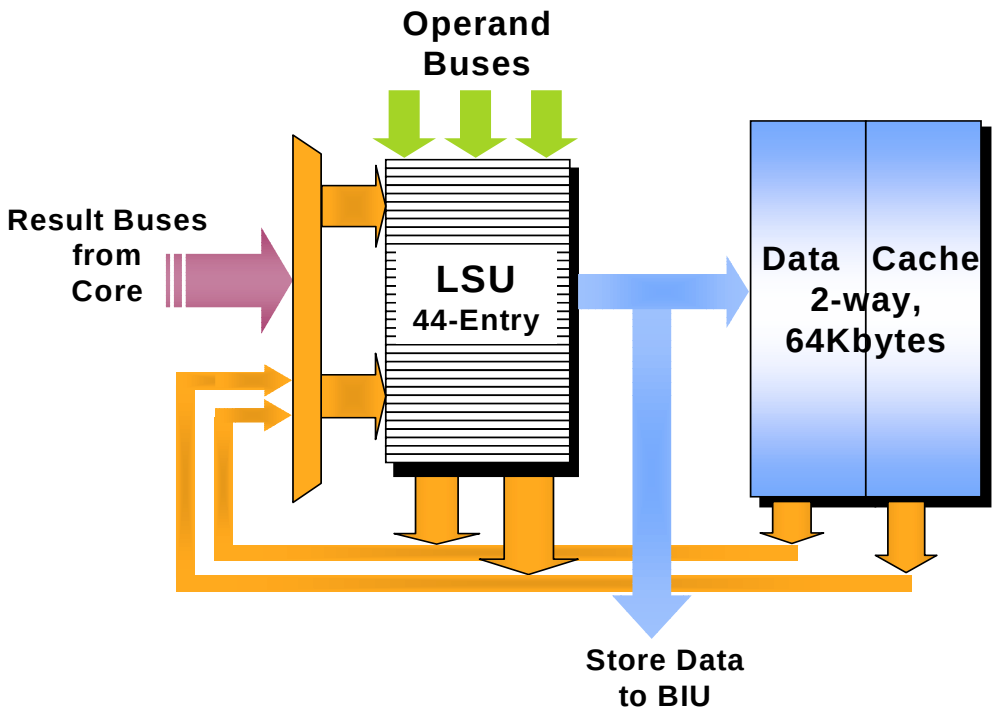

Figure 4. Load/Store Unit

L2 Cache Controller

The AMD Athlon processor contains a very flexible onboard L2 controller. It uses an independent backside bus to access up to 8-Mbytes of industry-standard SRAMs. There are full on-chip tags for a 512-Kbyte cache, while larger sizes use a partial tag system. In addition, there is a two-level data TLB structure. The first-level TLB is fully associative and contains 32 entries (24 that map 4-Kbyte pages and eight that map 2-Mbyte or 4-Mbyte pages). The second-level TLB is four-way set associative and contains 256 entries, which can map 4-Kbyte pages.

Write Combining

See Appendix C, “Implementation of Write Combining” on page 135 for detailed information about write combining.

AMD Athlon™ System Bus

The AMD Athlon system bus is a high-speed bus that consists of a pair of unidirectional 13-bit address and control channels and a bidirectional 64-bit data bus. The AMD Athlon system bus supports low-voltage swing, multiprocessing, clock forwarding, and fast data transfers. The clock forwarding technique is used to deliver data on both edges of the reference clock, therefore doubling the transfer speed. A four-entry 64-byte write buffer is integrated into the BIU. The write buffer improves bus utilization by combining multiple writes into a single large write cycle. By using the AMD Athlon system bus, the AMD Athlon processor can transfer data on the 64-bit data bus at 200 MHz, which yields an effective throughput of 1.6-Gbyte per second.

Appendix B

Pipeline and Execution Unit Resources Overview

The AMD Athlon™ processor contains two independent execution pipelines—one for integer operations and one for floating-point operations. The integer pipeline manages x86 integer operations and the floating-point pipeline manages all x87, 3DNow!™ and MMX™ instructions. This appendix describes the operation and functionality of these pipelines.

Fetch and Decode Pipeline Stages

Figure 5 on page 122 and Figure 6 on page 122 show the AMD Athlon processor instruction fetch and decoding pipeline stages. The pipeline consists of one cycle for instruction fetches and four cycles of instruction alignment and decoding. The three ports in stage 5 provide a maximum bandwidth of three MacroOPs per cycle for dispatching to the instruction control unit (ICU).

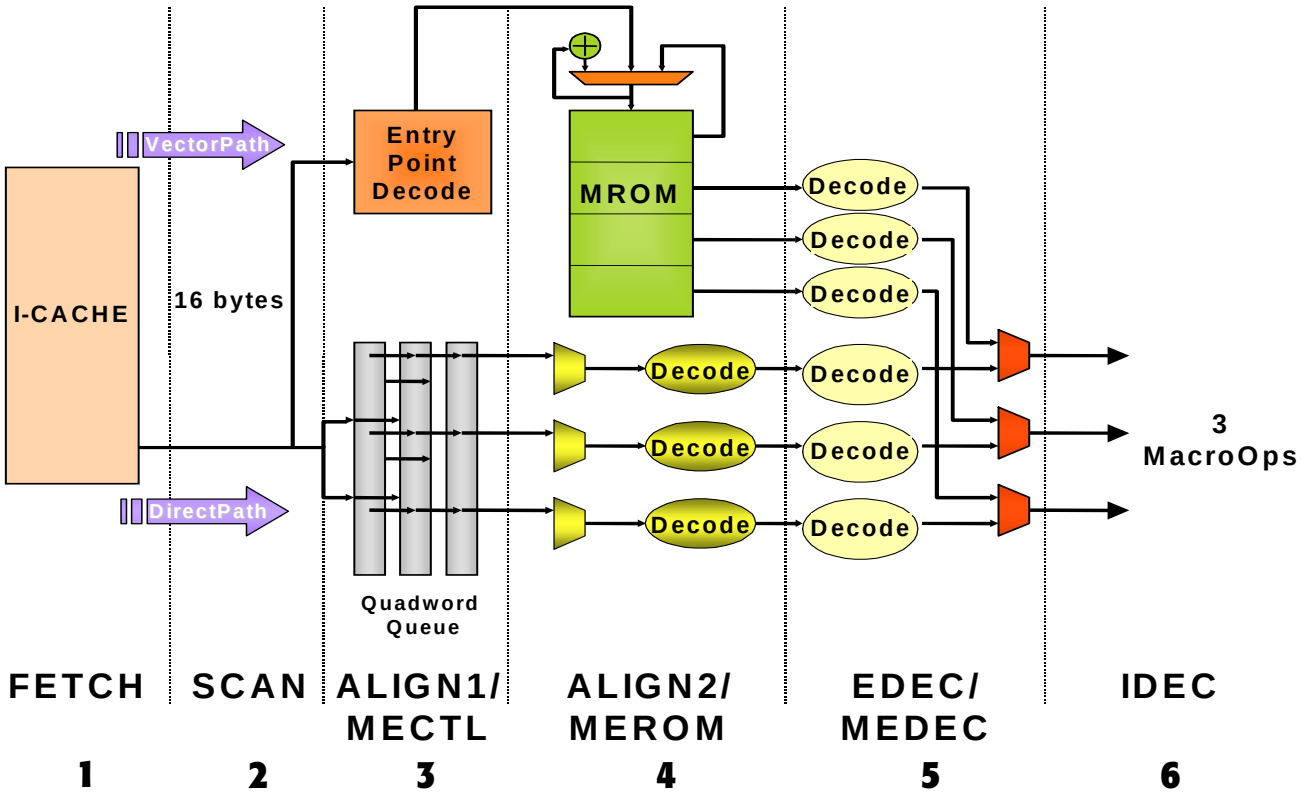

Figure 5. Fetch/Scan/Align/Decode Pipeline Hardware

The most common x86 instructions flow through the DirectPath pipeline stages and are decoded by hardware. The less common instructions, which require microcode assistance, flow through the VectorPath. Although the DirectPath decodes the common x86 instructions, it also contains VectorPath instruction data, which allows it to maintain dispatch order at the end of cycle 5.

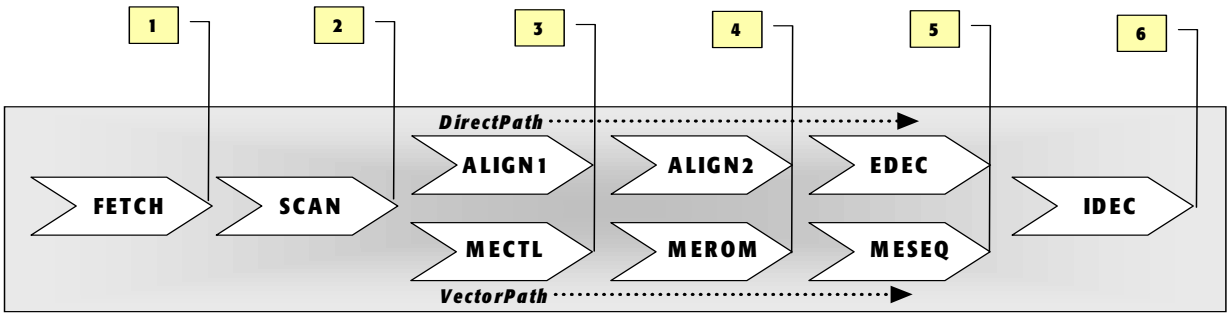

Figure 6. Fetch/Scan/Align/Decode Pipeline Stages

Cycle 1–FETCH The FETCH pipeline stage calculates the address of the next x86 instruction window to fetch from the processor caches or system memory.

Cycle 2–SCAN SCAN determines the start and end pointers of instructions. SCAN can send up to six *aligned* instructions (DirectPath and VectorPath) to ALIGN1 and only one VectorPath instruction to the microcode engine (MENG) per cycle.

Cycle 3 (DirectPath)–ALIGN1 Because each 8-byte buffer (quadword queue) can contain up to three instructions, ALIGN1 can buffer up to a maximum of nine instructions, or 24 instruction bytes. ALIGN1 tries to send three instructions from an 8-byte buffer to ALIGN2 per cycle.

Cycle 3 (VectorPath)–MECTL For VectorPath instructions, the microcode engine control (MECTL) stage of the pipeline generates the microcode entry points.

Cycle 4 (DirectPath)–ALIGN2 ALIGN2 prioritizes prefix bytes, determines the opcode, ModR/M, and SIB bytes for each instruction and sends the accumulated prefix information to EDEC.

Cycle 4 (VectorPath)–MEROM In the microcode engine ROM (MEROM) pipeline stage, the entry-point generated in the previous cycle, MECTL, is used to index into the MROM to obtain the microcode lines necessary to decode the instruction sent by SCAN.

Cycle 5 (DirectPath)–EDEC The early decode (EDEC) stage decodes information from the DirectPath stage (ALIGN2) and VectorPath stage (MEROM) into MacroOPs. In addition, EDEC determines register pointers, flag updates, immediate values, displacements, and other information. EDEC then selects either MacroOPs from the DirectPath or MacroOPs from the VectorPath to send to the instruction decoder (IDEC) stage.

Cycle 5 (VectorPath)–MEDEC/MESEQ The microcode engine decode (MEDEC) stage converts x86 instructions into MacroOPs. The microcode engine sequencer (MESEQ) performs the sequence controls (redirects and exceptions) for the MENG.

Cycle 6–IDEC/Rename At the instruction decoder (IDEC)/rename stage, integer and floating-point MacroOPs diverge in the pipeline. Integer MacroOPs are scheduled for execution in the next cycle. Floating-point MacroOPs have their floating-point stack

operands mapped to registers. Both integer and floating-point MacroOPs are placed into the ICU.

Integer Pipeline Stages

The integer execution pipeline consists of four or more stages for scheduling and execution and, if necessary, accessing data in the processor caches or system memory. There are three integer pipes associated with the three IEUs.

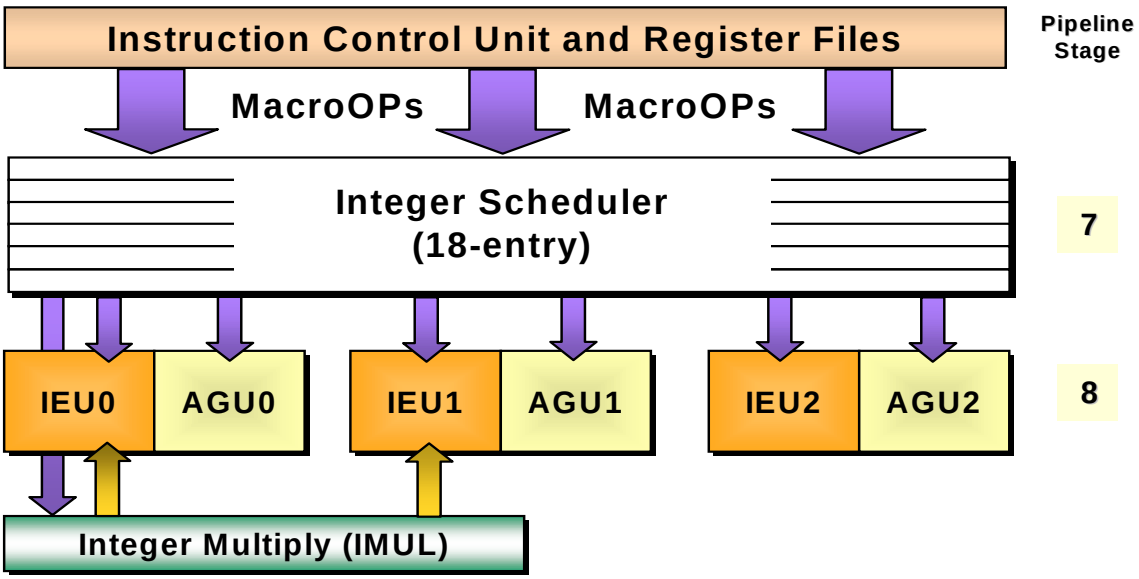

Figure 7. Integer Execution Pipeline

Figure 7 and Figure 8 show the integer execution resources and the pipeline stages, which are described in the following sections.

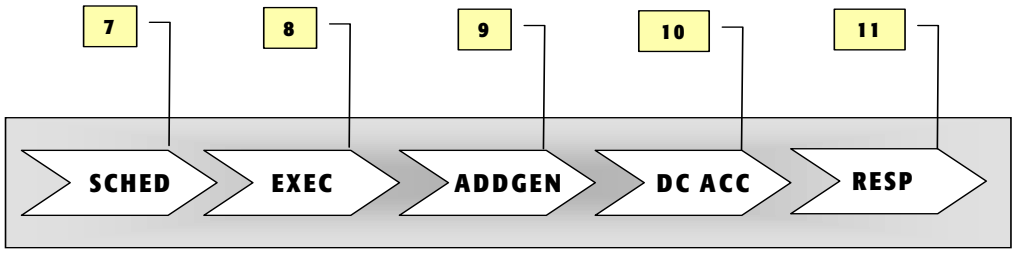

Figure 8. Integer Pipeline Stages

Cycle 7–SCHED In the scheduler (SCHED) pipeline stage, the scheduler buffers can contain MacroOPs that are waiting for integer operands from the ICU or the IEU result bus. When all operands are received, SCHED schedules the MacroOP for execution and issues the OPs to the next stage, EXEC.

Cycle 8–EXEC In the execution (EXEC) pipeline stage, the OP and its associated operands are processed by an integer pipe (either the IEU or the AGU). If addresses must be calculated to access data necessary to complete the operation, the OP proceeds to the next stages, ADDGEN and DCACC.

Cycle 9–ADDGEN In the address generation (ADDGEN) pipeline stage, the load or store OP calculates a linear address, which is sent to the data cache TLBs and caches.

Cycle 10–DCACC In the data cache access (DCACC) pipeline stage, the address generated in the previous pipeline stage is used to access the data cache arrays and TLBs. Any OP waiting in the scheduler for this data snarfs this data and proceeds to the EXEC stage (assuming all other operands were available).

Cycle 11–RESP In the response (RESP) pipeline stage, the data cache returns hit/miss status and data for the request from DCACC.

Floating-Point Pipeline Stages

The floating-point unit (FPU) is implemented as a coprocessor that has its own out-of-order control in addition to the data path. The FPU handles all register operations for x87 instructions, all 3DNow! operations, and all MMX operations. The FPU consists of a stack renaming unit, a register renaming unit, a scheduler, a register file, and three parallel execution units. Figure 9 shows a block diagram of the dataflow through the FPU.

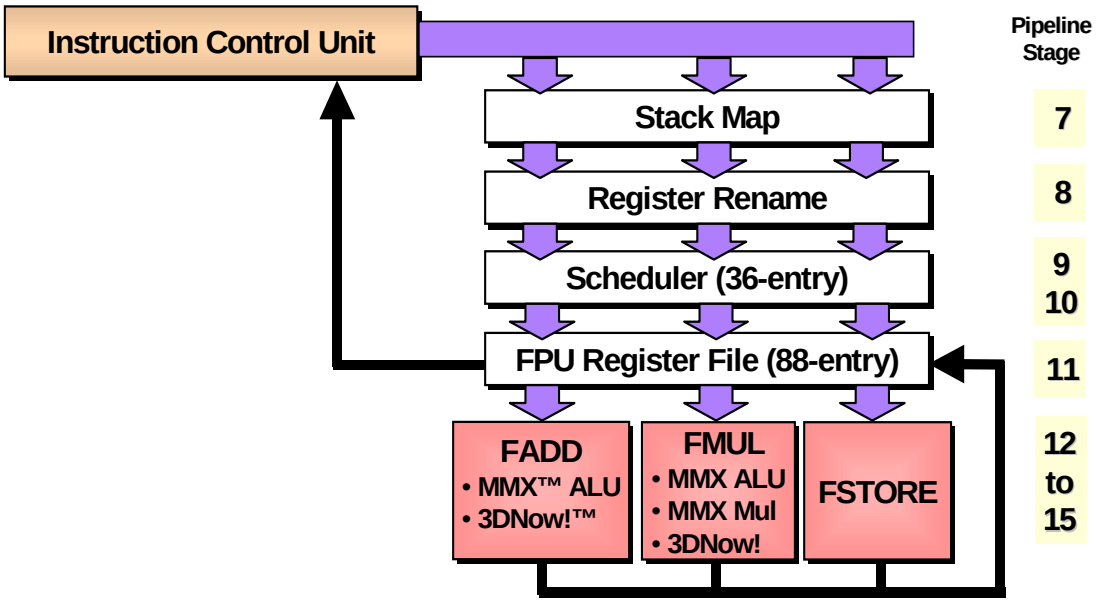

Figure 9. Floating-Point Unit Block Diagram

The floating-point pipeline stages 7–15 are shown in Figure 10 and described in the following sections. Note that the floating-point pipe and integer pipe separates at cycle 7.

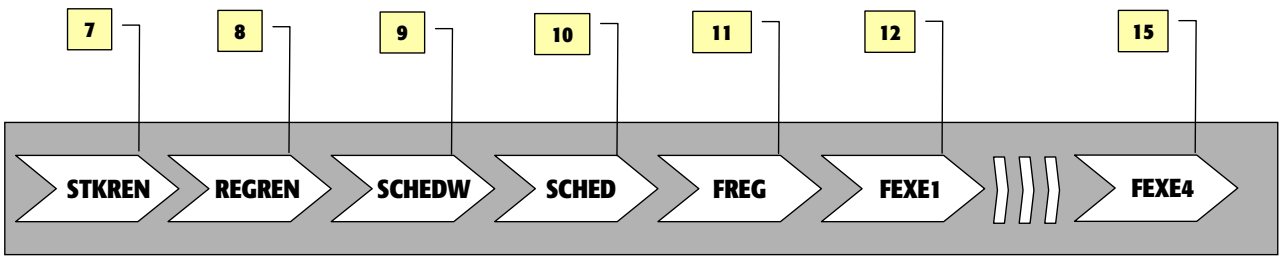

Figure 10. Floating-Point Pipeline Stages

Cycle 7–STKREN The stack rename (STKREN) pipeline stage in cycle 7 receives up to three MacroOPs from IDEC and maps stack-relative register tags to virtual register tags.

Cycle 8–REGREN The register renaming (REGREN) pipeline stage in cycle 8 is responsible for register renaming. In this stage, virtual register tags are mapped into physical register tags. Likewise, each destination is assigned a new physical register. The MacroOPs are then sent to the 36-entry FPU scheduler.

Cycle 9–SCHEDW The scheduler write (SCHEDW) pipeline stage in cycle 9 can receive up to three MacroOPs per cycle.

Cycle 10–SCHED The schedule (SCHED) pipeline stage in cycle 10 schedules up to three MacroOPs per cycle from the 36-entry FPU scheduler to the FREG pipeline stage to read register operands. MacroOPs are sent when their operands and/or tags are obtained.

Cycle 11–FREG The register file read (FREG) pipeline stage reads the floating-point register file for any register source operands of MacroOPs. The register file read is done before the MacroOPs are sent to the floating-point execution pipelines.

Cycle 12–15–Floating-Point Execution (FEXEC1–4) The FPU has three logical pipes—FADD, FMUL, and FSTORE. Each pipe may have several associated execution units. MMX execution is in both the FADD and FMUL pipes, with the exception of MMX instructions involving multiplies, which are limited to the FMUL pipe. The FMUL pipe has special support for long latency operations.

DirectPath/VectorPath operations are dispatched to the FPU during cycle 6, but are not acted upon until they receive validation from the ICU in cycle 7.

Execution Unit Resources

Terminology

The execution units operate with two types of register values—*operands* and *results*. There are three operand types and two result types, which are described in this section.

Operands

The three types of operands are as follows:

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

Results

The two types of results are as follows:

- *Data register results*—Produced by load or register instructions
- *Address register results*—Produced by LEA or PUSH instructions

Examples

The following examples illustrate the operand and result definitions:

```
ADD   EAX, EBX
```

The ADD instruction has two data register operands (EAX and EBX) and one data register result (EAX).

```
MOV   EBX, [ESP+4*ECX+8]      ;Load
```

The Load instruction has two address register operands (ESP and ECX as base and index registers, respectively) and a data register result (EBX).

```
MOV   [ESP+4*ECX+8], EAX      ;Store
```

The Store instruction has a data register operand (EAX) and two address register operands (ESP and ECX as base and index registers, respectively).

```
LEA   ESI, [ESP+4*ECX+8]
```

The LEA instruction has address register operands (ESP and ECX as base and index registers, respectively), and an address register result (ESI).

Integer Pipeline Operations

Table 2 shows the category or type of operations handled by the integer pipeline. Table 3 shows examples of the decode type.

Table 2. Integer Pipeline Operation Types

Category	Execution Unit
Integer Memory Load or Store Operations	L/S
Address Generation Operations	AGU
Integer Execution Unit Operations	IEU
Integer Multiply Operations	IMUL

Table 3. Integer Decode Types

x86 Instruction	Decode Type	OPs
MOV CX, [SP+4]	DirectPath	AGU, L/S
ADD AX, BX	DirectPath	IEU
CMP CX, [AX]	VectorPath	AGU, L/S, IEU
JZ Addr	DirectPath	IEU

As shown in Table 2, the MOV instruction early decodes in the DirectPath decoder and requires two OPs—an address generation operation for the indirect address and a data load from memory into a register. The ADD instruction early decodes in the DirectPath decoder and requires a single OP that can be executed in one of the three IEUs. The CMP instruction early decodes in the VectorPath and requires three OPs—an address generation operation for the indirect address, a data load from memory, and a compare to CX using an IEU. The final JZ instruction is a simple operation that early decodes in the DirectPath decoder and requires a single OP. Not shown is a load-op-store instruction, which translates into only one MacroOP (one AGU OP, one IEU OP, and one L/S OP).

Floating-Point Pipeline Operations

Table 4 shows the category or type of operations handled by the floating-point execution units. Table 5 shows examples of the decode types.

Table 4. Floating-Point Pipeline Operation Types

Category	Execution Unit
FPU/3DNow!/MMX Load/store or Miscellaneous Operations	FSTORE
FPU/3DNow!/MMX Multiply Operation	FMUL
FPU/3DNow!/MMX Arithmetic Operation	FADD

Table 5. Floating-Point Decode Types

x86 Instruction	Decode Type	OPs
FADD ST, ST(i)	DirectPath	FADD
FSIN	VectorPath	various
PFACC	DirectPath	FADD
PFRSQRT	DirectPath	FMUL

As shown in Table 4, the FADD register-to-register instruction generates a single MacroOP targeted for the floating-point scheduler. FSIN is considered a VectorPath instruction because it is a complex instruction with long execution times, as compared to the more common floating-point instructions. The MMX PFACC instruction is DirectPath decodeable and generates a single MacroOP targeted for the arithmetic operation execution pipeline in the floating-point logic. Just like PFACC, a single MacroOP is early decoded for the 3DNow! PFRSQRT instruction, but it is targeted for the multiply operation execution pipeline.

Load/Store Pipeline Operations

The AMD Athlon processor decodes any instruction that references memory into primitive load/store operations. For example, consider the following code sample:

```
MOV      AX, [EBX]          ;1 load MacroOP
PUSH     EAX                ;1 store MacroOP
POP      EAX                ;1 load MacroOP
ADD      [EAX], EBX         ;1 load/store and 1 IEU MacroOPs
FSTP     [EAX]              ;1 store MacroOP
MOVQ     [EAX], MM0         ;1 store MacroOP
```

As shown in Table 6, the load/store unit (LSU) consists of a three-stage data cache lookup.

Table 6. Load/Store Unit Stages

Stage 1 (Cycle 8)	Stage 2 (Cycle 9)	Stage 3 (Cycle 10)
Address Calculation / LS1 Scan	Transport Address to Data Cache	Data Cache Access / LS2 Data Forward

Loads and stores are first dispatched in order into a 12-entry deep reservation queue called LS1. LS1 holds loads and stores that are waiting to enter the cache subsystem. Loads and stores are allocated into LS1 entries at dispatch time in program order, and are required by LS1 to probe the data cache in program order. The AGUs can calculate addresses out of program order, therefore, LS1 acts as an address reorder buffer.

When a load or store is scanned out of the LS1 queue (Stage 1), it is deallocated from the LS1 queue and inserted into the data cache probe pipeline (Stage 2 and Stage 3). Up to two memory operations can be scheduled (scanned out of LS1) to access the data cache per cycle. The LSU can handle the following:

- Two 64-bit loads per cycle or
- One 64-bit load and one 64-bit store per cycle or
- Two 32-bit stores per cycle

Code Sample Analysis

The samples in Table 7 on page 133 and Table 8 on page 134 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 decode pipe in the integer execution pipeline, the decode type, the clock counts, and a description of the events occurring within the processor. The decode pipe gives the specific IEU used (see Figure 7 on page 124). The decode type specifies either VectorPath (VP) or DirectPath (DP).

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

- D—Dispatch stage (Allocate in ICU, reservation stations, load-store (LS1) queue)
- I—Issue stage (Schedule operation for AGU or FU execution)
- E—Integer Execution Unit (IEU number corresponds to decode pipe)
- &—Address Generation Unit (AGU number corresponds to decode pipe)
- M—Multiplier Execution
- S—Load/Store pipe stage 1 (Schedule operation for load/store pipe)
- A—Load/Store pipe stage 2 (1st stage of data cache/LS2 buffer access)
- $—Load/Store pipe stage 3 (2nd stage of data cache/LS2 buffer access)

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 7. Sample 1 – Integer Register Operations

Instruction Number	Instruction	Decode Pipe	Decode Type	Clocks							
				1	2	3	4	5	6	7	8
1	IMUL EAX, ECX	0	VP	D	I	M	M	M	M		
2	INC ESI	0	DP		D	I	E				
3	MOV EDI, 0x07F4	1	DP		D	I	E				
4	ADD EDI, EBX	2	DP		D		I	E			
5	SHL EAX, 8	0	DP			D			I	E	
6	OR EAX, 0x0F	1	DP			D				I	E
7	INC EBX	2	DP			D		I	E		
8	ADD ESI, EDX	0	DP				D	I	E		

Comments for Each Instruction Number

1. The IMUL is a VectorPath instruction. It cannot be decode or paired with other operations and, therefore, dispatches alone in pipe 0. The multiply latency is four cycles.
2. The simple INC operation is paired with instructions 3 and 4. The INC executes in IEU0 in cycle 4.
3. The MOV executes in IEU1 in cycle 4.
4. The ADD operation depends on instruction 3. It executes in IEU2 in cycle 5.
5. The SHL operation depends on the multiply result (instruction 1). The MacroOP waits in a reservation station and is eventually scheduled to execute in cycle 7 after the multiply result is available.
6. This operation executes in cycle 8 in IEU1.
7. This simple operation has a resource contention for execution in IEU2 in cycle 5. Therefore, the operation does not execute until cycle 6.
8. The ADD operation executes immediately in IEU0 after dispatching.

Table 8. Sample 2 – Integer Register and Memory Load Operations

Instruc Num	Instruction	Decode Pipe	Decode Type	Clocks											
				1	2	3	4	5	6	7	8	9	10	11	12
1	DEC EDX	0	DP	D	I	E									
2	MOV EDI, [ECX]	1	DP	D	I	&/S	A	$							
3	SUB EAX, [EDX+20]	2	DP	D	I	&/S	A	$/I	E						
4	SAR EAX, 5	0	DP		D				I	E					
5	ADD ECX, [EDI+4]	1	DP		D			I	&/S	A	$				
6	AND EBX, 0x1F	2	DP		D	I	E								
7	MOV ESI, [0x0F100]	0	DP			D	I	&	S	A	$				
8	OR ECX, [ESI+EAX*4+8]	1	DP			D					I	&/S	A	$	E

Comments for Each Instruction Number

1. The ALU operation executes in IEU0.
2. The load operation generates the address in AGU1 and is simultaneously scheduled for the load/store pipe in cycle 3. In cycles 4 and 5, the load completes the data cache access.
3. The load-execute instruction accesses the data cache in tandem with instruction 2. After the load portion completes, the subtraction is executed in cycle 6 in IEU2.
4. The shift operation executes in IEU0 (cycle 7) after instruction 3 completes.
5. This operation is stalled on its address calculation waiting for instruction 2 to update EDI. The address is calculated in cycle 6. In cycle 7/8, the cache access completes.
6. This simple operation executes quickly in IEU2
7. The address for the load is calculated in cycle 5 in AGU0. However, the load is not scheduled to access the data cache until cycle 6. The load is blocked for scheduling to access the data cache for one cycle by instruction 5. In cycles 7 and 8, instruction 7 accesses the data cache concurrently with instruction 5.
8. The load execute instruction accesses the data cache in cycles 10/11 and executes the 'OR' operation in IEU1 in cycle 12.

Appendix C

Implementation of Write Combining

Introduction

This appendix describes the memory write-combining feature as implemented in the AMD Athlon™ processor family. The AMD Athlon processor supports the memory type and range register (MTRR) and the page attribute table (PAT) extensions, which allow software to define ranges of memory as either writeback (WB), write-protected (WP), writethrough (WT), uncacheable (UC), or write-combining (WC).

Defining the memory type for a range of memory as WC or WT allows the processor to conditionally combine data from multiple write cycles that are addressed within this range into a merge buffer. Merging multiple write cycles into a single write cycle reduces processor bus utilization and processor stalls, thereby increasing the overall system performance.

To understand the information presented in this appendix, the reader should possess a knowledge of K86™ processors, the x86 architecture, and programming requirements.

Write-Combining Definitions and Abbreviations

This appendix uses the following definitions and abbreviations.

- UC—Uncacheable memory type
- WC—Write-combining memory type
- WT—Writethrough memory type
- WP—Write-protected memory type
- WB—Writeback memory type
- One Byte—8 bits
- One Word—16 bits
- Longword—32 bits (same as a x86 doubleword)
- Quadword—64 bits or 2 longwords
- Octaword—128 bits or 2 quadwords
- Cache Block—64 bytes or 4 octawords or 8 quadwords

What is Write Combining?

Write combining is the merging of multiple memory write cycles that target locations within the address range of a write buffer. The AMD Athlon processor combines multiple memory-write cycles to a 64-byte buffer whenever the memory address is within a WC or WT memory type region. The processor continues to combine writes to this buffer without writing the data to the system, as long as certain rules apply (see Table 9 on page 138 for more information).

Programming Details

The steps required for programming write combining on the AMD Athlon processor are as follows:

1. Verify the presence of an AMD Athlon processor by using the CPUID instruction to check for the instruction family code and vendor identification of the processor. Standard function 0 on AMD processors returns a vendor identification string of “AuthenticAMD” in registers EBX, EDX, and ECX. Standard function 1 returns the processor

signature in register EAX, where EAX[11–8] contains the instruction family code. For the AMD Athlon processor, the instruction family code is six (6).

2. In addition, the presence of the MTRRs is indicated by bit 12 and the presence of the PAT extension is indicated by bit 16 of the extended features bits returned in the EDX register by CPUID function 8000_0001h. See the *AMD Processor Recognition Application Note, order# 20734* for more details on the CPUID instruction.
3. Write combining is controlled by the MTRRs and PAT. Write combining should be enabled for the appropriate memory ranges. The AMD Athlon processor MTRRs and PAT are compatible with the Pentium® II.

Write-Combining Operations

In order to improve system performance, the AMD Athlon processor aggressively combines multiple memory-write cycles of any data size that address locations within a 64-byte write buffer that is aligned to a cache-line boundary. The data sizes can be bytes, words, longwords, or quadwords.

WC memory type writes can be combined in any order up to a full 64-byte sized write buffer.

WT memory type writes can only be combined up to a fully aligned quadword in the 64-byte buffer, and must be combined contiguously in ascending order. Combining may be opened at any byte boundary in a quadword, but is closed by a write that is either not “contiguous and ascending” or fills byte 7.

All other memory types for stores that go through the write buffer (UC and WP) cannot be combined.

Combining is able to continue until interrupted by one of the conditions listed in Table 9 on page 138. When combining is interrupted, one or more bus commands are issued to the system for that write buffer, as described by Table 10 on page 139.

Table 9. Write Combining Completion Events

Event	Comment
Non-WB write outside of current buffer	The first non-WB write to a different cache block address closes combining for previous writes. WB writes do not affect write combining. Only one line-sized buffer can be open for write combining at a time. Once a buffer is closed for write combining, it cannot be reopened for write combining.
I/O Read or Write	Any IN/INS or OUT/OUTS instruction closes combining. The implied memory type for all IN/OUT instructions is UC, which cannot be combined.
Serializing instructions	Any serializing instruction closes combining. These instructions include: MOVCRx, MOVDRx, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, LTR, CPUID, IRET, RSM, INIT, HALT.
Flushing instructions	Any flush instruction causes the WC to complete.
Locks	Any instruction or processor operation that requires a cache or bus lock closes write combining before starting the lock. Writes within a lock can be combined.
Uncacheable Read	A UC read closes write combining. A WC read closes combining only if a cache block address match occurs between the WC read and a write in the write buffer.
Different memory type	Any WT write while write-combining for WC memory or any WC write while write combining for WT memory closes write combining.
Buffer full	Write combining is closed if all 64 bytes of the write buffer are valid.
WT time-out	If 16 processor clocks have passed since the most recent write for WT write combining, write combining is closed. There is no time-out for WC write combining.
WT write fills byte 7	Write combining is closed if a write fills the most significant byte of a quadword, which includes writes that are misaligned across a quadword boundary. In the misaligned case, combining is closed by the LS part of the misaligned write and combining is opened by the MS part of the misaligned store.
WT Nonsequential	If a subsequent WT write is not in ascending sequential order, the write combining completes. WC writes have no addressing constraints within the 64-byte line being combined.
TLB AD bit set	Write combining is closed whenever a TLB reload sets the accessed (A) or dirty (D) bits of a Pde or Pte.

Sending Write-Buffer Data to the System

Once write combining is closed for a 64-byte write buffer, the contents of the write buffer are eligible to be sent to the system as one or more AMD Athlon system bus commands. Table 10 lists the rules for determining what system commands are issued for a write buffer, as a function of the alignment of the valid buffer data.

Table 10. AMD Athlon™ System Bus Commands Generation Rules

1.	If all eight quadwords are either full (8 bytes valid) or empty (0 bytes valid), a Write-Quadword system command is issued, with an 8-byte mask representing which of the eight quadwords are valid. If this case is true, do not proceed to the next rule.
2.	If all longwords are either full (4 bytes valid) or empty (0 bytes valid), a Write-Longword system command is issued for each 32-byte buffer half that contains at least one valid longword. The mask for each Write-Longword system command indicates which longwords are valid in that 32-byte write buffer half. If this case is true, do not proceed to the next rule.
3.	Sequence through all eight quadwords of the write buffer, from quadword 0 to quadword 7. Skip over a quadword if no bytes are valid. Issue a Write-Quad system command if all bytes are valid, asserting one mask bit. Issue a Write-Longword system command if the quadword contains one aligned longword, asserting one mask bit. Otherwise, issue a Write-Byte system command if there is at least one valid byte, asserting a mask bit for each valid byte.

Appendix D

Instruction Dispatch and Execution Timing

This chapter describes the MacroOPs generated by each decoded instruction, along with the relative static execution latencies of these groups of operations. Tables 11 through 16 starting on page 142 define the integer, MMX™, MMX extensions, floating-point, 3DNow!™, and 3DNow! extensions instructions, respectively.

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/3DNow! register
- *mmreg1*—MMX/3DNow! register defined by bits 5, 4, and 3 of the modR/M byte
- *mmreg2*—MMX/3DNow! register defined by bits 2, 1, and 0 of the modR/M byte

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

The fourth column lists the modR/M byte 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—DirectPath or VectorPath (see "DirectPath Decoder" on page 113 and "VectorPath Decoder" on page 113 for more information). The AMD Athlon™ enhanced processor decode logic can process three instructions per clock.

The FPU, MMX, and 3DNow! instruction tables have an additional column that lists the possible FPU execution pipelines available for use by any particular DirectPath decoded operation. Typically, VectorPath instructions require more than one execution pipe resource.

Table 11. Integer Instructions

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
AAA	37h			VectorPath
AAD	D5h	0Ah		VectorPath
AAM	D4h	0Ah		VectorPath
AAS	3Fh			VectorPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
ADC mreg8, reg8	10h		11-xxx-xxx	DirectPath
ADC mem8, reg8	10h		mm-xxx-xxx	DirectPath
ADC mreg16/32, reg16/32	11h		11-xxx-xxx	DirectPath
ADC mem16/32, reg16/32	11h		mm-xxx-xxx	DirectPath
ADC reg8, mreg8	12h		11-xxx-xxx	DirectPath
ADC reg8, mem8	12h		mm-xxx-xxx	DirectPath
ADC reg16/32, mreg16/32	13h		11-xxx-xxx	DirectPath
ADC reg16/32, mem16/32	13h		mm-xxx-xxx	DirectPath
ADC AL, imm8	14h			DirectPath
ADC EAX, imm16/32	15h			DirectPath
ADC mreg8, imm8	80h		11-010-xxx	DirectPath
ADC mem8, imm8	80h		mm-010-xxx	DirectPath
ADC mreg16/32, imm16/32	81h		11-010-xxx	DirectPath
ADC mem16/32, imm16/32	81h		mm-010-xxx	DirectPath
ADC mreg16/32, imm8 (sign extended)	83h		11-010-xxx	DirectPath
ADC mem16/32, imm8 (sign extended)	83h		mm-010-xxx	DirectPath
ADD mreg8, reg8	00h		11-xxx-xxx	DirectPath
ADD mem8, reg8	00h		mm-xxx-xxx	DirectPath
ADD mreg16/32, reg16/32	01h		11-xxx-xxx	DirectPath
ADD mem16/32, reg16/32	01h		mm-xxx-xxx	DirectPath
ADD reg8, mreg8	02h		11-xxx-xxx	DirectPath
ADD reg8, mem8	02h		mm-xxx-xxx	DirectPath
ADD reg16/32, mreg16/32	03h		11-xxx-xxx	DirectPath
ADD reg16/32, mem16/32	03h		mm-xxx-xxx	DirectPath
ADD AL, imm8	04h			DirectPath
ADD EAX, imm16/32	05h			DirectPath
ADD mreg8, imm8	80h		11-000-xxx	DirectPath
ADD mem8, imm8	80h		mm-000-xxx	DirectPath
ADD mreg16/32, imm16/32	81h		11-000-xxx	DirectPath
ADD mem16/32, imm16/32	81h		mm-000-xxx	DirectPath
ADD mreg16/32, imm8 (sign extended)	83h		11-000-xxx	DirectPath
ADD mem16/32, imm8 (sign extended)	83h		mm-000-xxx	DirectPath
AND mreg8, reg8	20h		11-xxx-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
AND mem8, reg8	20h		mm-xxx-xxx	DirectPath
AND mreg16/32, reg16/32	21h		11-xxx-xxx	DirectPath
AND mem16/32, reg16/32	21h		mm-xxx-xxx	DirectPath
AND reg8, mreg8	22h		11-xxx-xxx	DirectPath
AND reg8, mem8	22h		mm-xxx-xxx	DirectPath
AND reg16/32, mreg16/32	23h		11-xxx-xxx	DirectPath
AND reg16/32, mem16/32	23h		mm-xxx-xxx	DirectPath
AND AL, imm8	24h			DirectPath
AND EAX, imm16/32	25h			DirectPath
AND mreg8, imm8	80h		11-100-xxx	DirectPath
AND mem8, imm8	80h		mm-100-xxx	DirectPath
AND mreg16/32, imm16/32	81h		11-100-xxx	DirectPath
AND mem16/32, imm16/32	81h		mm-100-xxx	DirectPath
AND mreg16/32, imm8 (sign extended)	83h		11-100-xxx	DirectPath
AND mem16/32, imm8 (sign extended)	83h		mm-100-xxx	DirectPath
ARPL mreg16, reg16	63h		11-xxx-xxx	VectorPath
ARPL mem16, reg16	63h		mm-xxx-xxx	VectorPath
BOUND	62h			VectorPath
BSF reg16/32, mreg16/32	0Fh	BCh	11-xxx-xxx	VectorPath
BSF reg16/32, mem16/32	0Fh	BCh	mm-xxx-xxx	VectorPath
BSR reg16/32, mreg16/32	0Fh	BDh	11-xxx-xxx	VectorPath
BSR reg16/32, mem16/32	0Fh	BDh	mm-xxx-xxx	VectorPath
BSWAP EAX	0Fh	C8h		DirectPath
BSWAP ECX	0Fh	C9h		DirectPath
BSWAP EDX	0Fh	CAh		DirectPath
BSWAP EBX	0Fh	CBh		DirectPath
BSWAP ESP	0Fh	CCh		DirectPath
BSWAP EBP	0Fh	CDh		DirectPath
BSWAP ESI	0Fh	CEh		DirectPath
BSWAP EDI	0Fh	CFh		DirectPath
BT mreg16/32, reg16/32	0Fh	A3h	11-xxx-xxx	DirectPath
BT mem16/32, reg16/32	0Fh	A3h	mm-xxx-xxx	VectorPath
BT mreg16/32, imm8	0Fh	BAh	11-100-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
BT mem16/32, imm8	0Fh	BAh	mm-100-xxx	DirectPath
BTC mreg16/32, reg16/32	0Fh	BBh	11-xxx-xxx	VectorPath
BTC mem16/32, reg16/32	0Fh	BBh	mm-xxx-xxx	VectorPath
BTC mreg16/32, imm8	0Fh	BAh	11-111-xxx	VectorPath
BTC mem16/32, imm8	0Fh	BAh	mm-111-xxx	VectorPath
BTR mreg16/32, reg16/32	0Fh	B3h	11-xxx-xxx	VectorPath
BTR mem16/32, reg16/32	0Fh	B3h	mm-xxx-xxx	VectorPath
BTR mreg16/32, imm8	0Fh	BAh	11-110-xxx	VectorPath
BTR mem16/32, imm8	0Fh	BAh	mm-110-xxx	VectorPath
BTS mreg16/32, reg16/32	0Fh	ABh	11-xxx-xxx	VectorPath
BTS mem16/32, reg16/32	0Fh	ABh	mm-xxx-xxx	VectorPath
BTS mreg16/32, imm8	0Fh	BAh	11-101-xxx	VectorPath
BTS mem16/32, imm8	0Fh	BAh	mm-101-xxx	VectorPath
CALL full pointer	9Ah			VectorPath
CALL near imm16/32	E8h			VectorPath
CALL mem16:16/32	FFh		11-011-xxx	VectorPath
CALL near mreg32 (indirect)	FFh		11-010-xxx	VectorPath
CALL near mem32 (indirect)	FFh		mm-010-xxx	VectorPath
CBW/CWDE	98h			DirectPath
CLC	F8h			DirectPath
CLD	FCh			VectorPath
CLI	FAh			VectorPath
CLTS	0Fh	06h		VectorPath
CMC	F5h			DirectPath
CMOVA/CMOVNBE reg16/32, reg16/32	0Fh	47h	11-xxx-xxx	DirectPath
CMOVA/CMOVNBE reg16/32, mem16/32	0Fh	47h	mm-xxx-xxx	DirectPath
CMOVAE/CMOVNB/CMOVNC reg16/32, mem16/32	0Fh	43h	11-xxx-xxx	DirectPath
CMOVAE/CMOVNB/CMOVNC mem16/32, mem16/32	0Fh	43h	mm-xxx-xxx	DirectPath
CMOVB/CMOVC/CMOVNAE reg16/32, reg16/32	0Fh	42h	11-xxx-xxx	DirectPath
CMOVB/CMOVC/CMOVNAE mem16/32, reg16/32	0Fh	42h	mm-xxx-xxx	DirectPath
CMOVBE/CMOVNA reg16/32, reg16/32	0Fh	46h	11-xxx-xxx	DirectPath
CMOVBE/CMOVNA reg16/32, mem16/32	0Fh	46h	mm-xxx-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
CMOVE/CMOVZ reg16/32, reg16/32	0Fh	44h	11-xxx-xxx	DirectPath
CMOVE/CMOVZ reg16/32, mem16/32	0Fh	44h	mm-xxx-xxx	DirectPath
CMOVG/CMOVNLE reg16/32, reg16/32	0Fh	4Fh	11-xxx-xxx	DirectPath
CMOVG/CMOVNLE reg16/32, mem16/32	0Fh	4Fh	mm-xxx-xxx	DirectPath
CMOVGE/CMOVNL reg16/32, reg16/32	0Fh	4Dh	11-xxx-xxx	DirectPath
CMOVGE/CMOVNL reg16/32, mem16/32	0Fh	4Dh	mm-xxx-xxx	DirectPath
CMOVL/CMOVNGE reg16/32, reg16/32	0Fh	4Ch	11-xxx-xxx	DirectPath
CMOVL/CMOVNGE reg16/32, mem16/32	0Fh	4Ch	mm-xxx-xxx	DirectPath
CMOVLE/CMOVNG reg16/32, reg16/32	0Fh	4Eh	11-xxx-xxx	DirectPath
CMOVLE/CMOVNG reg16/32, mem16/32	0Fh	4Eh	mm-xxx-xxx	DirectPath
CMOVNE/CMOVNZ reg16/32, reg16/32	0Fh	45h	11-xxx-xxx	DirectPath
CMOVNE/CMOVNZ reg16/32, mem16/32	0Fh	45h	mm-xxx-xxx	DirectPath
CMOVNO reg16/32, reg16/32	0Fh	41h	11-xxx-xxx	DirectPath
CMOVNO reg16/32, mem16/32	0Fh	41h	mm-xxx-xxx	DirectPath
CMOVNP/CMOVPO reg16/32, reg16/32	0Fh	4Bh	11-xxx-xxx	DirectPath
CMOVNP/CMOVPO reg16/32, mem16/32	0Fh	4Bh	mm-xxx-xxx	DirectPath
CMOVNS reg16/32, reg16/32	0Fh	49h	11-xxx-xxx	DirectPath
CMOVNS reg16/32, mem16/32	0Fh	49h	mm-xxx-xxx	DirectPath
CMOVO reg16/32, reg16/32	0Fh	40h	11-xxx-xxx	DirectPath
CMOVO reg16/32, mem16/32	0Fh	40h	mm-xxx-xxx	DirectPath
CMOVP/CMOVPE reg16/32, reg16/32	0Fh	4Ah	11-xxx-xxx	DirectPath
CMOVP/CMOVPE reg16/32, mem16/32	0Fh	4Ah	mm-xxx-xxx	DirectPath
CMOVS reg16/32, reg16/32	0Fh	48h	11-xxx-xxx	DirectPath
CMOVS reg16/32, mem16/32	0Fh	48h	mm-xxx-xxx	DirectPath
CMP mreg8, reg8	38h		11-xxx-xxx	DirectPath
CMP mem8, reg8	38h		mm-xxx-xxx	DirectPath
CMP mreg16/32, reg16/32	39h		11-xxx-xxx	DirectPath
CMP mem16/32, reg16/32	39h		mm-xxx-xxx	DirectPath
CMP reg8, mreg8	3Ah		11-xxx-xxx	DirectPath
CMP reg8, mem8	3Ah		mm-xxx-xxx	DirectPath
CMP reg16/32, mreg16/32	3Bh		11-xxx-xxx	DirectPath
CMP reg16/32, mem16/32	3Bh		mm-xxx-xxx	DirectPath
CMP AL, imm8	3Ch			DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
CMP EAX, imm16/32	3Dh			DirectPath
CMP mreg8, imm8	80h		11-111-xxx	DirectPath
CMP mem8, imm8	80h		mm-111-xxx	DirectPath
CMP mreg16/32, imm16/32	81h		11-111-xxx	DirectPath
CMP mem16/32, imm16/32	81h		mm-111-xxx	DirectPath
CMP mreg16/32, imm8 (sign extended)	83h		11-111-xxx	DirectPath
CMP mem16/32, imm8 (sign extended)	83h		mm-111-xxx	DirectPath
CMPSB mem8,mem8	A6h			VectorPath
CMPSW mem16, mem32	A7h			VectorPath
CMPSD mem32, mem32	A7h			VectorPath
CMPXCHG mreg8, reg8	0Fh	B0h	11-xxx-xxx	VectorPath
CMPXCHG mem8, reg8	0Fh	B0h	mm-xxx-xxx	VectorPath
CMPXCHG mreg16/32, reg16/32	0Fh	B1h	11-xxx-xxx	VectorPath
CMPXCHG mem16/32, reg16/32	0Fh	B1h	mm-xxx-xxx	VectorPath
CMPXCHG8B mem64	0Fh	C7h	mm-xxx-xxx	VectorPath
CPUID	0Fh	A2h		VectorPath
CWD/CDQ	99h			DirectPath
DAA	27h			VectorPath
DAS	2Fh			VectorPath
DEC EAX	48h			DirectPath
DEC ECX	49h			DirectPath
DEC EDX	4Ah			DirectPath
DEC EBX	4Bh			DirectPath
DEC ESP	4Ch			DirectPath
DEC EBP	4Dh			DirectPath
DEC ESI	4Eh			DirectPath
DEC EDI	4Fh			DirectPath
DEC mreg8	FEh		11-001-xxx	DirectPath
DEC mem8	FEh		mm-001-xxx	DirectPath
DEC mreg16/32	FFh		11-001-xxx	DirectPath
DEC mem16/32	FFh		mm-001-xxx	DirectPath
DIV AL, mreg8	F6h		11-110-xxx	VectorPath
DIV AL, mem8	F6h		mm-110-xxx	VectorPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
DIV EAX, mreg16/32	F7h		11-110-xxx	VectorPath
DIV EAX, mem16/32	F7h		mm-110-xxx	VectorPath
ENTER	C8			VectorPath
IDIV mreg8	F6h		11-111-xxx	VectorPath
IDIV mem8	F6h		mm-111-xxx	VectorPath
IDIV EAX, mreg16/32	F7h		11-111-xxx	VectorPath
IDIV EAX, mem16/32	F7h		mm-111-xxx	VectorPath
IMUL reg16/32, imm16/32	69h		11-xxx-xxx	VectorPath
IMUL reg16/32, mreg16/32, imm16/32	69h		11-xxx-xxx	VectorPath
IMUL reg16/32, mem16/32, imm16/32	69h		mm-xxx-xxx	VectorPath
IMUL reg16/32, imm8 (sign extended)	6Bh		11-xxx-xxx	VectorPath
IMUL reg16/32, mreg16/32, imm8 (signed)	6Bh		11-xxx-xxx	VectorPath
IMUL reg16/32, mem16/32, imm8 (signed)	6Bh		mm-xxx-xxx	VectorPath
IMUL AX, AL, mreg8	F6h		11-101-xxx	VectorPath
IMUL AX, AL, mem8	F6h		mm-101-xxx	VectorPath
IMUL EDX:EAX, EAX, mreg16/32	F7h		11-101-xxx	VectorPath
IMUL EDX:EAX, EAX, mem16/32	F7h		mm-101-xxx	VectorPath
IMUL reg16/32, mreg16/32	0Fh	AFh	11-xxx-xxx	VectorPath
IMUL reg16/32, mem16/32	0Fh	AFh	mm-xxx-xxx	VectorPath
IN AL, imm8	E4h			VectorPath
IN AX, imm8	E5h			VectorPath
IN EAX, imm8	E5h			VectorPath
IN AL, DX	ECh			VectorPath
IN AX, DX	EDh			VectorPath
IN EAX, DX	EDh			VectorPath
INC EAX	40h			DirectPath
INC ECX	41h			DirectPath
INC EDX	42h			DirectPath
INC EBX	43h			DirectPath
INC ESP	44h			DirectPath
INC EBP	45h			DirectPath
INC ESI	46h			DirectPath
INC EDI	47h			DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
INC mreg8	FEh		11-000-xxx	DirectPath
INC mem8	FEh		mm-000-xxx	DirectPath
INC mreg16/32	FFh		11-000-xxx	DirectPath
INC mem16/32	FFh		mm-000-xxx	DirectPath
INVD	0Fh	08h		VectorPath
INVLPG	0Fh	01h	mm-111-xxx	VectorPath
JO short disp8	70h			DirectPath
JNO short disp8	71h			DirectPath
JB/JNAE/JC short disp8	72h			DirectPath
JNB/JAE/JNC short disp8	73h			DirectPath
JZ/JE short disp8	74h			DirectPath
JNZ/JNE short disp8	75h			DirectPath
JBE/JNA short disp8	76h			DirectPath
JNBE/JA short disp8	77h			DirectPath
JS short disp8	78h			DirectPath
JNS short disp8	79h			DirectPath
JP/JPE short disp8	7Ah			DirectPath
JNP/JPO short disp8	7Bh			DirectPath
JL/JNGE short disp8	7Ch			DirectPath
JNL/JGE short disp8	7Dh			DirectPath
JLE/JNG short disp8	7Eh			DirectPath
JNLE/JG short disp8	7Fh			DirectPath
JCXZ/JEC short disp8	E3h			VectorPath
JO near disp16/32	0Fh	80h		DirectPath
JNO near disp16/32	0Fh	81h		DirectPath
JB/JNAE near disp16/32	0Fh	82h		DirectPath
JNB/JAE near disp16/32	0Fh	83h		DirectPath
JZ/JE near disp16/32	0Fh	84h		DirectPath
JNZ/JNE near disp16/32	0Fh	85h		DirectPath
JBE/JNA near disp16/32	0Fh	86h		DirectPath
JNBE/JA near disp16/32	0Fh	87h		DirectPath
JS near disp16/32	0Fh	88h		DirectPath
JNS near disp16/32	0Fh	89h		DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
JP/JPE near disp16/32	0Fh	8Ah		DirectPath
JNP/JPO near disp16/32	0Fh	8Bh		DirectPath
JL/JNGE near disp16/32	0Fh	8Ch		DirectPath
JNL/JGE near disp16/32	0Fh	8Dh		DirectPath
JLE/JNG near disp16/32	0Fh	8Eh		DirectPath
JNLE/JG near disp16/32	0Fh	8Fh		DirectPath
JMP near disp16/32 (direct)	E9h			DirectPath
JMP far disp32/48 (direct)	EAh			VectorPath
JMP disp8 (short)	EBh			DirectPath
JMP far mem32 (indirect)	EFh		mm-101-xxx	VectorPath
JMP far mreg32 (indirect)	FFh		mm-101-xxx	VectorPath
JMP near mreg16/32 (indirect)	FFh		11-100-xxx	DirectPath
JMP near mem16/32 (indirect)	FFh		mm-100-xxx	DirectPath
LAHF	9Fh			VectorPath
LAR reg16/32, mreg16/32	0Fh	02h	11-xxx-xxx	VectorPath
LAR reg16/32, mem16/32	0Fh	02h	mm-xxx-xxx	VectorPath
LDS reg16/32, mem32/48	C5h		mm-xxx-xxx	VectorPath
LEA reg16, mem16/32	8Dh		mm-xxx-xxx	VectorPath
LEA reg32, mem16/32	8Dh		mm-xxx-xxx	DirectPath
LEAVE	C9h			VectorPath
LES reg16/32, mem32/48	C4h		mm-xxx-xxx	VectorPath
LFS reg16/32, mem32/48	0Fh	B4h		VectorPath
LGDT mem48	0Fh	01h	mm-010-xxx	VectorPath
LGS reg16/32, mem32/48	0Fh	B5h		VectorPath
LIDT mem48	0Fh	01h	mm-011-xxx	VectorPath
LLDT mreg16	0Fh	00h	11-010-xxx	VectorPath
LLDT mem16	0Fh	00h	mm-010-xxx	VectorPath
LMSW mreg16	0Fh	01h	11-100-xxx	VectorPath
LMSW mem16	0Fh	01h	mm-100-xxx	VectorPath
LODSB AL, mem8	ACh			VectorPath
LODSW AX, mem16	ADh			VectorPath
LODSD EAX, mem32	ADh			VectorPath
LOOP disp8	E2h			VectorPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
LOOPE/LOOPZ disp8	E1h			VectorPath
LOOPNE/LOOPNZ disp8	E0h			VectorPath
LSL reg16/32, mreg16/32	0Fh	03h	11-xxx-xxx	VectorPath
LSL reg16/32, mem16/32	0Fh	03h	mm-xxx-xxx	VectorPath
LSS reg16/32, mem32/48	0Fh	B2h	mm-xxx-xxx	VectorPath
LTR mreg16	0Fh	00h	11-011-xxx	VectorPath
LTR mem16	0Fh	00h	mm-011-xxx	VectorPath
MOV mreg8, reg8	88h		11-xxx-xxx	DirectPath
MOV mem8, reg8	88h		mm-xxx-xxx	DirectPath
MOV mreg16/32, reg16/32	89h		11-xxx-xxx	DirectPath
MOV mem16/32, reg16/32	89h		mm-xxx-xxx	DirectPath
MOV reg8, mreg8	8Ah		11-xxx-xxx	DirectPath
MOV reg8, mem8	8Ah		mm-xxx-xxx	DirectPath
MOV reg16/32, mreg16/32	8Bh		11-xxx-xxx	DirectPath
MOV reg16/32, mem16/32	8Bh		mm-xxx-xxx	DirectPath
MOV mreg16, segment reg	8Ch		11-xxx-xxx	VectorPath
MOV mem16, segment reg	8Ch		mm-xxx-xxx	VectorPath
MOV segment reg, mreg16	8Eh		11-xxx-xxx	VectorPath
MOV segment reg, mem16	8Eh		mm-xxx-xxx	VectorPath
MOV AL, mem8	A0h			DirectPath
MOV EAX, mem16/32	A1h			DirectPath
MOV mem8, AL	A2h			DirectPath
MOV mem16/32, EAX	A3h			DirectPath
MOV AL, imm8	B0h			DirectPath
MOV CL, imm8	B1h			DirectPath
MOV DL, imm8	B2h			DirectPath
MOV BL, imm8	B3h			DirectPath
MOV AH, imm8	B4h			DirectPath
MOV CH, imm8	B5h			DirectPath
MOV DH, imm8	B6h			DirectPath
MOV BH, imm8	B7h			DirectPath
MOV EAX, imm16/32	B8h			DirectPath
MOV ECX, imm16/32	B9h			DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
MOV EDX, imm16/32	BAh			DirectPath
MOV EBX, imm16/32	BBh			DirectPath
MOV ESP, imm16/32	BCh			DirectPath
MOV EBP, imm16/32	BDh			DirectPath
MOV ESI, imm16/32	BEh			DirectPath
MOV EDI, imm16/32	BFh			DirectPath
MOV mreg8, imm8	C6h		11-000-xxx	DirectPath
MOV mem8, imm8	C6h		mm-000-xxx	DirectPath
MOV mreg16/32, imm16/32	C7h		11-000-xxx	DirectPath
MOV mem16/32, imm16/32	C7h		mm-000-xxx	DirectPath
MOVSB mem8,mem8	A4h			VectorPath
MOVSD mem16, mem16	A5h			VectorPath
MOVSW mem32, mem32	A5h			VectorPath
MOVSX reg16/32, mreg8	0Fh	BEh	11-xxx-xxx	DirectPath
MOVSX reg16/32, mem8	0Fh	BEh	mm-xxx-xxx	DirectPath
MOVSX reg32, mreg16	0Fh	BFh	11-xxx-xxx	DirectPath
MOVSX reg32, mem16	0Fh	BFh	mm-xxx-xxx	DirectPath
MOVZX reg16/32, mreg8	0Fh	B6h	11-xxx-xxx	DirectPath
MOVZX reg16/32, mem8	0Fh	B6h	mm-xxx-xxx	DirectPath
MOVZX reg32, mreg16	0Fh	B7h	11-xxx-xxx	DirectPath
MOVZX reg32, mem16	0Fh	B7h	mm-xxx-xxx	DirectPath
MUL AL, mreg8	F6h		11-100-xxx	VectorPath
MUL AL, mem8	F6h		mm-100-xx	VectorPath
MUL AX, mreg16	F7h		11-100-xxx	VectorPath
MUL AX, mem16	F7h		mm-100-xxx	VectorPath
MUL EAX, mreg32	F7h		11-100-xxx	VectorPath
MUL EAX, mem32	F7h		mm-100-xx	VectorPath
NEG mreg8	F6h		11-011-xxx	DirectPath
NEG mem8	F6h		mm-011-xx	DirectPath
NEG mreg16/32	F7h		11-011-xxx	DirectPath
NEG mem16/32	F7h		mm-011-xx	DirectPath
NOP (XCHG EAX, EAX)	90h			DirectPath
NOT mreg8	F6h		11-010-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
NOT mem8	F6h		mm-010-xx	DirectPath
NOT mreg16/32	F7h		11-010-xxx	DirectPath
NOT mem16/32	F7h		mm-010-xx	DirectPath
OR mreg8, reg8	08h		11-xxx-xxx	DirectPath
OR mem8, reg8	08h		mm-xxx-xxx	DirectPath
OR mreg16/32, reg16/32	09h		11-xxx-xxx	DirectPath
OR mem16/32, reg16/32	09h		mm-xxx-xxx	DirectPath
OR reg8, mreg8	0Ah		11-xxx-xxx	DirectPath
OR reg8, mem8	0Ah		mm-xxx-xxx	DirectPath
OR reg16/32, mreg16/32	0Bh		11-xxx-xxx	DirectPath
OR reg16/32, mem16/32	0Bh		mm-xxx-xxx	DirectPath
OR AL, imm8	0Ch			DirectPath
OR EAX, imm16/32	0Dh			DirectPath
OR mreg8, imm8	80h		11-001-xxx	DirectPath
OR mem8, imm8	80h		mm-001-xxx	DirectPath
OR mreg16/32, imm16/32	81h		11-001-xxx	DirectPath
OR mem16/32, imm16/32	81h		mm-001-xxx	DirectPath
OR mreg16/32, imm8 (sign extended)	83h		11-001-xxx	DirectPath
OR mem16/32, imm8 (sign extended)	83h		mm-001-xxx	DirectPath
OUT imm8, AL	E6h			VectorPath
OUT imm8, AX	E7h			VectorPath
OUT imm8, EAX	E7h			VectorPath
OUT DX, AL	EEh			VectorPath
OUT DX, AX	EFh			VectorPath
OUT DX, EAX	EFh			VectorPath
POP ES	07h			VectorPath
POP SS	17h			VectorPath
POP DS	1Fh			VectorPath
POP FS	0Fh	A1h		VectorPath
POP GS	0Fh	A9h		VectorPath
POP EAX	58h			VectorPath
POP ECX	59h			VectorPath
POP EDX	5Ah			VectorPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
POP EBX	5Bh			VectorPath
POP ESP	5Ch			VectorPath
POP EBP	5Dh			VectorPath
POP ESI	5Eh			VectorPath
POP EDI	5Fh			VectorPath
POP mreg 16/32	8Fh		11-000-xxx	VectorPath
POP mem 16/32	8Fh		mm-000-xxx	VectorPath
POPA/POPAD	61h			VectorPath
POPF/POPFD	9Dh			VectorPath
PUSH ES	06h			VectorPath
PUSH CS	0Eh			VectorPath
PUSH FS	0Fh	A0h		VectorPath
PUSH GS	0Fh	A8h		VectorPath
PUSH SS	16h			VectorPath
PUSH DS	1Eh			VectorPath
PUSH EAX	50h			DirectPath
PUSH ECX	51h			DirectPath
PUSH EDX	52h			DirectPath
PUSH EBX	53h			DirectPath
PUSH ESP	54h			DirectPath
PUSH EBP	55h			DirectPath
PUSH ESI	56h			DirectPath
PUSH EDI	57h			DirectPath
PUSH imm8	6Ah			DirectPath
PUSH imm16/32	68h			DirectPath
PUSH mreg16/32	FFh		11-110-xxx	VectorPath
PUSH mem16/32	FFh		mm-110-xxx	VectorPath
PUSHA/PUSHAD	60h			VectorPath
PUSHF/PUSHFD	9Ch			VectorPath
RCL mreg8, imm8	C0h		11-010-xxx	DirectPath
RCL mem8, imm8	C0h		mm-010-xxx	VectorPath
RCL mreg16/32, imm8	C1h		11-010-xxx	DirectPath
RCL mem16/32, imm8	C1h		mm-010-xxx	VectorPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
RCL mreg8, 1	D0h		11-010-xxx	DirectPath
RCL mem8, 1	D0h		mm-010-xxx	DirectPath
RCL mreg16/32, 1	D1h		11-010-xxx	DirectPath
RCL mem16/32, 1	D1h		mm-010-xxx	DirectPath
RCL mreg8, CL	D2h		11-010-xxx	DirectPath
RCL mem8, CL	D2h		mm-010-xxx	VectorPath
RCL mreg16/32, CL	D3h		11-010-xxx	DirectPath
RCL mem16/32, CL	D3h		mm-010-xxx	VectorPath
RCR mreg8, imm8	C0h		11-011-xxx	DirectPath
RCR mem8, imm8	C0h		mm-011-xxx	VectorPath
RCR mreg16/32, imm8	C1h		11-011-xxx	DirectPath
RCR mem16/32, imm8	C1h		mm-011-xxx	VectorPath
RCR mreg8, 1	D0h		11-011-xxx	DirectPath
RCR mem8, 1	D0h		mm-011-xxx	DirectPath
RCR mreg16/32, 1	D1h		11-011-xxx	DirectPath
RCR mem16/32, 1	D1h		mm-011-xxx	DirectPath
RCR mreg8, CL	D2h		11-011-xxx	DirectPath
RCR mem8, CL	D2h		mm-011-xxx	VectorPath
RCR mreg16/32, CL	D3h		11-011-xxx	DirectPath
RCR mem16/32, CL	D3h		mm-011-xxx	VectorPath
RDMSR	0Fh	32h		VectorPath
RDPMC	0Fh	33h		VectorPath
RDTSC	0F	31h		VectorPath
RET near imm16	C2h			VectorPath
RET near	C3h			VectorPath
RET far imm16	CAh			VectorPath
RET far	CBh			VectorPath
ROL mreg8, imm8	C0h		11-000-xxx	DirectPath
ROL mem8, imm8	C0h		mm-000-xxx	DirectPath
ROL mreg16/32, imm8	C1h		11-000-xxx	DirectPath
ROL mem16/32, imm8	C1h		mm-000-xxx	DirectPath
ROL mreg8, 1	D0h		11-000-xxx	DirectPath
ROL mem8, 1	D0h		mm-000-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
ROL mreg16/32, 1	D1h		11-000-xxx	DirectPath
ROL mem16/32, 1	D1h		mm-000-xxx	DirectPath
ROL mreg8, CL	D2h		11-000-xxx	DirectPath
ROL mem8, CL	D2h		mm-000-xxx	DirectPath
ROL mreg16/32, CL	D3h		11-000-xxx	DirectPath
ROL mem16/32, CL	D3h		mm-000-xxx	DirectPath
ROR mreg8, imm8	C0h		11-001-xxx	DirectPath
ROR mem8, imm8	C0h		mm-001-xxx	DirectPath
ROR mreg16/32, imm8	C1h		11-001-xxx	DirectPath
ROR mem16/32, imm8	C1h		mm-001-xxx	DirectPath
ROR mreg8, 1	D0h		11-001-xxx	DirectPath
ROR mem8, 1	D0h		mm-001-xxx	DirectPath
ROR mreg16/32, 1	D1h		11-001-xxx	DirectPath
ROR mem16/32, 1	D1h		mm-001-xxx	DirectPath
ROR mreg8, CL	D2h		11-001-xxx	DirectPath
ROR mem8, CL	D2h		mm-001-xxx	DirectPath
ROR mreg16/32, CL	D3h		11-001-xxx	DirectPath
ROR mem16/32, CL	D3h		mm-001-xxx	DirectPath
SAHF	9Eh			VectorPath
SAR mreg8, imm8	C0h		11-111-xxx	DirectPath
SAR mem8, imm8	C0h		mm-111-xxx	DirectPath
SAR mreg16/32, imm8	C1h		11-111-xxx	DirectPath
SAR mem16/32, imm8	C1h		mm-111-xxx	DirectPath
SAR mreg8, 1	D0h		11-111-xxx	DirectPath
SAR mem8, 1	D0h		mm-111-xxx	DirectPath
SAR mreg16/32, 1	D1h		11-111-xxx	DirectPath
SAR mem16/32, 1	D1h		mm-111-xxx	DirectPath
SAR mreg8, CL	D2h		11-111-xxx	DirectPath
SAR mem8, CL	D2h		mm-111-xxx	DirectPath
SAR mreg16/32, CL	D3h		11-111-xxx	DirectPath
SAR mem16/32, CL	D3h		mm-111-xxx	DirectPath
SBB mreg8, reg8	18h		11-xxx-xxx	DirectPath
SBB mem8, reg8	18h		mm-xxx-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
SBB mreg16/32, reg16/32	19h		11-xxx-xxx	DirectPath
SBB mem16/32, reg16/32	19h		mm-xxx-xxx	DirectPath
SBB reg8, mreg8	1Ah		11-xxx-xxx	DirectPath
SBB reg8, mem8	1Ah		mm-xxx-xxx	DirectPath
SBB reg16/32, mreg16/32	1Bh		11-xxx-xxx	DirectPath
SBB reg16/32, mem16/32	1Bh		mm-xxx-xxx	DirectPath
SBB AL, imm8	1Ch			DirectPath
SBB EAX, imm16/32	1Dh			DirectPath
SBB mreg8, imm8	80h		11-011-xxx	DirectPath
SBB mem8, imm8	80h		mm-011-xxx	DirectPath
SBB mreg16/32, imm16/32	81h		11-011-xxx	DirectPath
SBB mem16/32, imm16/32	81h		mm-011-xxx	DirectPath
SBB mreg16/32, imm8 (sign extended)	83h		11-011-xxx	DirectPath
SBB mem16/32, imm8 (sign extended)	83h		mm-011-xxx	DirectPath
SCASB AL, mem8	AEh			VectorPath
SCASW AX, mem16	AFh			VectorPath
SCASD EAX, mem32	AFh			VectorPath
SETO mreg8	0Fh	90h	11-xxx-xxx	DirectPath
SETO mem8	0Fh	90h	mm-xxx-xxx	DirectPath
SETNO mreg8	0Fh	91h	11-xxx-xxx	DirectPath
SETNO mem8	0Fh	91h	mm-xxx-xxx	DirectPath
SETB/SETC/SETNAE mreg8	0Fh	92h	11-xxx-xxx	DirectPath
SETB/SETC/SETNAE mem8	0Fh	92h	mm-xxx-xxx	DirectPath
SETAE/SETNB/SETNC mreg8	0Fh	93h	11-xxx-xxx	DirectPath
SETAE/SETNB/SETNC mem8	0Fh	93h	mm-xxx-xxx	DirectPath
SETE/SETZ mreg8	0Fh	94h	11-xxx-xxx	DirectPath
SETE/SETZ mem8	0Fh	94h	mm-xxx-xxx	DirectPath
SETNE/SETNZ mreg8	0Fh	95h	11-xxx-xxx	DirectPath
SETNE/SETNZ mem8	0Fh	95h	mm-xxx-xxx	DirectPath
SETBE/SETNA mreg8	0Fh	96h	11-xxx-xxx	DirectPath
SETBE/SETNA mem8	0Fh	96h	mm-xxx-xxx	DirectPath
SETA/SETNBE mreg8	0Fh	97h	11-xxx-xxx	DirectPath
SETA/SETNBE mem8	0Fh	97h	mm-xxx-xxx	DirectPath

Table 11. Integer Instructions (continued)

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

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
SHR mem16/32, imm8	C1h		mm-101-xxx	DirectPath
SHR mreg8, 1	D0h		11-101-xxx	DirectPath
SHR mem8, 1	D0h		mm-101-xxx	DirectPath
SHR mreg16/32, 1	D1h		11-101-xxx	DirectPath
SHR mem16/32, 1	D1h		mm-101-xxx	DirectPath
SHR mreg8, CL	D2h		11-101-xxx	DirectPath
SHR mem8, CL	D2h		mm-101-xxx	DirectPath
SHR mreg16/32, CL	D3h		11-101-xxx	DirectPath
SHR mem16/32, CL	D3h		mm-101-xxx	DirectPath
SHLD mreg16/32, reg16/32, imm8	0Fh	A4h	11-xxx-xxx	VectorPath
SHLD mem16/32, reg16/32, imm8	0Fh	A4h	mm-xxx-xxx	VectorPath
SHLD mreg16/32, reg16/32, CL	0Fh	A5h	11-xxx-xxx	VectorPath
SHLD mem16/32, reg16/32, CL	0Fh	A5h	mm-xxx-xxx	VectorPath
SHRD mreg16/32, reg16/32, imm8	0Fh	ACh	11-xxx-xxx	VectorPath
SHRD mem16/32, reg16/32, imm8	0Fh	ACh	mm-xxx-xxx	VectorPath
SHRD mreg16/32, reg16/32, CL	0Fh	ADh	11-xxx-xxx	VectorPath
SHRD mem16/32, reg16/32, CL	0Fh	ADh	mm-xxx-xxx	VectorPath
SLDT mreg16	0Fh	00h	11-000-xxx	VectorPath
SLDT mem16	0Fh	00h	mm-000-xxx	VectorPath
SMSW mreg16	0Fh	01h	11-100-xxx	VectorPath
SMSW mem16	0Fh	01h	mm-100-xxx	VectorPath
STC	F9h			DirectPath
STD	FDh			VectorPath
STI	FBh			VectorPath
STOSB mem8, AL	AAh			VectorPath
STOSW mem16, AX	ABh			VectorPath
STOSD mem32, EAX	ABh			VectorPath
STR mreg16	0Fh	00h	11-001-xxx	VectorPath
STR mem16	0Fh	00h	mm-001-xxx	VectorPath
SUB mreg8, reg8	28h		11-xxx-xxx	DirectPath
SUB mem8, reg8	28h		mm-xxx-xxx	DirectPath
SUB mreg16/32, reg16/32	29h		11-xxx-xxx	DirectPath
SUB mem16/32, reg16/32	29h		mm-xxx-xxx	DirectPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
SUB reg8, mreg8	2Ah		11-xxx-xxx	DirectPath
SUB reg8, mem8	2Ah		mm-xxx-xxx	DirectPath
SUB reg16/32, mreg16/32	2Bh		11-xxx-xxx	DirectPath
SUB reg16/32, mem16/32	2Bh		mm-xxx-xxx	DirectPath
SUB AL, imm8	2Ch			DirectPath
SUB EAX, imm16/32	2Dh			DirectPath
SUB mreg8, imm8	80h		11-101-xxx	DirectPath
SUB mem8, imm8	80h		mm-101-xxx	DirectPath
SUB mreg16/32, imm16/32	81h		11-101-xxx	DirectPath
SUB mem16/32, imm16/32	81h		mm-101-xxx	DirectPath
SUB mreg16/32, imm8 (sign extended)	83h		11-101-xxx	DirectPath
SUB mem16/32, imm8 (sign extended)	83h		mm-101-xxx	DirectPath
SYSCALL	0Fh	05h		VectorPath
SYSENTER	0Fh	34h		VectorPath
SYSEXIT	0Fh	35h		VectorPath
SYSRET	0Fh	07h		VectorPath
TEST mreg8, reg8	84h		11-xxx-xxx	DirectPath
TEST mem8, reg8	84h		mm-xxx-xxx	DirectPath
TEST mreg16/32, reg16/32	85h		11-xxx-xxx	DirectPath
TEST mem16/32, reg16/32	85h		mm-xxx-xxx	DirectPath
TEST AL, imm8	A8h			DirectPath
TEST EAX, imm16/32	A9h			DirectPath
TEST mreg8, imm8	F6h		11-000-xxx	DirectPath
TEST mem8, imm8	F6h		mm-000-xxx	DirectPath
TEST mreg8, imm16/32	F7h		11-000-xxx	DirectPath
TEST mem8, imm16/32	F7h		mm-000-xxx	DirectPath
VERR mreg16	0Fh	00h	11-100-xxx	VectorPath
VERR mem16	0Fh	00h	mm-100-xxx	VectorPath
VERW mreg16	0Fh	00h	11-101-xxx	VectorPath
VERW mem16	0Fh	00h	mm-101-xxx	VectorPath
WAIT	9Bh			DirectPath
WBINVD	0Fh	09h		VectorPath
WRMSR	0Fh	30h		VectorPath

Table 11. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type
XADD mreg8, reg8	0Fh	C0h	11-100-xxx	VectorPath
XADD mem8, reg8	0Fh	C0h	mm-100-xxx	VectorPath
XADD mreg16/32, reg16/32	0Fh	C1h	11-101-xxx	VectorPath
XADD mem16/32, reg16/32	0Fh	C1h	mm-101-xxx	VectorPath
XCHG reg8, mreg8	86h		11-xxx-xxx	VectorPath
XCHG reg8, mem8	86h		mm-xxx-xxx	VectorPath
XCHG reg16/32, mreg16/32	87h		11-xxx-xxx	VectorPath
XCHG reg16/32, mem16/32	87h		mm-xxx-xxx	VectorPath
XCHG EAX, EAX	90h			DirectPath
XCHG EAX, ECX	91h			VectorPath
XCHG EAX, EDX	92h			VectorPath
XCHG EAX, EBX	93h			VectorPath
XCHG EAX, ESP	94h			VectorPath
XCHG EAX, EBP	95h			VectorPath
XCHG EAX, ESI	96h			VectorPath
XCHG EAX, EDI	97h			VectorPath
XLAT	D7h			VectorPath
XOR mreg8, reg8	30h		11-xxx-xxx	DirectPath
XOR mem8, reg8	30h		mm-xxx-xxx	DirectPath
XOR mreg16/32, reg16/32	31h		11-xxx-xxx	DirectPath
XOR mem16/32, reg16/32	31h		mm-xxx-xxx	DirectPath
XOR reg8, mreg8	32h		11-xxx-xxx	DirectPath
XOR reg8, mem8	32h		mm-xxx-xxx	DirectPath
XOR reg16/32, mreg16/32	33h		11-xxx-xxx	DirectPath
XOR reg16/32, mem16/32	33h		mm-xxx-xxx	DirectPath
XOR AL, imm8	34h			DirectPath
XOR EAX, imm16/32	35h			DirectPath
XOR mreg8, imm8	80h		11-110-xxx	DirectPath
XOR mem8, imm8	80h		mm-110-xxx	DirectPath
XOR mreg16/32, imm16/32	81h		11-110-xxx	DirectPath
XOR mem16/32, imm16/32	81h		mm-110-xxx	DirectPath
XOR mreg16/32, imm8 (sign extended)	83h		11-110-xxx	DirectPath
XOR mem16/32, imm8 (sign extended)	83h		mm-110-xxx	DirectPath

Table 12. MMX™ Instructions

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Notes
EMMS	0Fh	77h		DirectPath	FADD/FMUL/FSTORE	
MOVD mmreg, reg32	0Fh	6Eh	11-xxx-xxx	VectorPath		1
MOVD mmreg, mem32	0Fh	6Eh	mm-xxx-xxx	DirectPath	FADD/FMUL/FSTORE	
MOVD reg32, mmreg	0Fh	7Eh	11-xxx-xxx	VectorPath		1
MOVD mem32, mmreg	0Fh	7Eh	mm-xxx-xxx	DirectPath	FSTORE	
MOVQ mmreg1, mmreg2	0Fh	6Fh	11-xxx-xxx	DirectPath	FADD/FMUL	
MOVQ mmreg, mem64	0Fh	6Fh	mm-xxx-xxx	DirectPath	FADD/FMUL/FSTORE	
MOVQ mmreg2, mmreg1	0Fh	7Fh	11-xxx-xxx	DirectPath	FADD/FMUL	
MOVQ mem64, mmreg	0Fh	7Fh	mm-xxx-xxx	DirectPath	FSTORE	
PACKSSDW mmreg1, mmreg2	0Fh	6Bh	11-xxx-xxx	DirectPath	FADD/FMUL	
PACKSSDW mmreg, mem64	0Fh	6Bh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PACKSSWB mmreg1, mmreg2	0Fh	63h	11-xxx-xxx	DirectPath	FADD/FMUL	
PACKSSWB mmreg, mem64	0Fh	63h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PACKUSWB mmreg1, mmreg2	0Fh	67h	11-xxx-xxx	DirectPath	FADD/FMUL	
PACKUSWB mmreg, mem64	0Fh	67h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDB mmreg1, mmreg2	0Fh	FCh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDB mmreg, mem64	0Fh	FCh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDD mmreg1, mmreg2	0Fh	FEh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDD mmreg, mem64	0Fh	FEh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDSB mmreg1, mmreg2	0Fh	ECh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDSB mmreg, mem64	0Fh	ECh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDSW mmreg1, mmreg2	0Fh	EDh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDSW mmreg, mem64	0Fh	EDh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDUSB mmreg1, mmreg2	0Fh	DCh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDUSB mmreg, mem64	0Fh	DCh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDUSW mmreg1, mmreg2	0Fh	DDh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDUSW mmreg, mem64	0Fh	DDh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PADDW mmreg1, mmreg2	0Fh	FDh	11-xxx-xxx	DirectPath	FADD/FMUL	
PADDW mmreg, mem64	0Fh	FDh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PAND mmreg1, mmreg2	0Fh	DBh	11-xxx-xxx	DirectPath	FADD/FMUL	
PAND mmreg, mem64	0Fh	DBh	mm-xxx-xxx	DirectPath	FADD/FMUL	

Notes:

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

Table 12. MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Notes
PANDN mmreg1, mmreg2	0Fh	DFh	11-xxx-xxx	DirectPath	FADD/FMUL	
PANDN mmreg, mem64	0Fh	DFh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PCMPEQB mmreg1, mmreg2	0Fh	74h	11-xxx-xxx	DirectPath	FADD/FMUL	
PCMPEQB mmreg, mem64	0Fh	74h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PCMPEQD mmreg1, mmreg2	0Fh	76h	11-xxx-xxx	DirectPath	FADD/FMUL	
PCMPEQD mmreg, mem64	0Fh	76h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PCMPEQW mmreg1, mmreg2	0Fh	75h	11-xxx-xxx	DirectPath	FADD/FMUL	
PCMPEQW mmreg, mem64	0Fh	75h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PCMPGTB mmreg1, mmreg2	0Fh	64h	11-xxx-xxx	DirectPath	FADD/FMUL	
PCMPGTB mmreg, mem64	0Fh	64h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PCMPGTD mmreg1, mmreg2	0Fh	66h	11-xxx-xxx	DirectPath	FADD/FMUL	
PCMPGTD mmreg, mem64	0Fh	66h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PCMPGTW mmreg1, mmreg2	0Fh	65h	11-xxx-xxx	DirectPath	FADD/FMUL	
PCMPGTW mmreg, mem64	0Fh	65h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PMADDWD mmreg1, mmreg2	0Fh	F5h	11-xxx-xxx	DirectPath	FMUL	
PMADDWD mmreg, mem64	0Fh	F5h	mm-xxx-xxx	DirectPath	FMUL	
PMULHW mmreg1, mmreg2	0Fh	E5h	11-xxx-xxx	DirectPath	FMUL	
PMULHW mmreg, mem64	0Fh	E5h	mm-xxx-xxx	DirectPath	FMUL	
PMULLW mmreg1, mmreg2	0Fh	D5h	11-xxx-xxx	DirectPath	FMUL	
PMULLW mmreg, mem64	0Fh	D5h	mm-xxx-xxx	DirectPath	FMUL	
POR mmreg1, mmreg2	0Fh	EBh	11-xxx-xxx	DirectPath	FADD/FMUL	
POR mmreg, mem64	0Fh	EBh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSLLD mmreg1, mmreg2	0Fh	F2h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSLLD mmreg, mem64	0Fh	F2h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSLLD mmreg, imm8	0Fh	72h	11-110-xxx	DirectPath	FADD/FMUL	
PSLLQ mmreg1, mmreg2	0Fh	F3h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSLLQ mmreg, mem64	0Fh	F3h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSLLQ mmreg, imm8	0Fh	73h	11-110-xxx	DirectPath	FADD/FMUL	
PSLLW mmreg1, mmreg2	0Fh	F1h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSLLW mmreg, mem64	0Fh	F1h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSLLW mmreg, imm8	0Fh	71h	11-110-xxx	DirectPath	FADD/FMUL	

Notes:

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

Table 12. MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Notes
PSRAW mmreg1, mmreg2	0Fh	E1h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSRAW mmreg, mem64	0Fh	E1h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSRAW mmreg, imm8	0Fh	71h	11-100-xxx	DirectPath	FADD/FMUL	
PSRAD mmreg1, mmreg2	0Fh	E2h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSRAD mmreg, mem64	0Fh	E2h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSRAD mmreg, imm8	0Fh	72h	11-100-xxx	DirectPath	FADD/FMUL	
PSRLD mmreg1, mmreg2	0Fh	D2h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSRLD mmreg, mem64	0Fh	D2h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSRLD mmreg, imm8	0Fh	72h	11-010-xxx	DirectPath	FADD/FMUL	
PSRLQ mmreg1, mmreg2	0Fh	D3h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSRLQ mmreg, mem64	0Fh	D3h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSRLQ mmreg, imm8	0Fh	73h	11-010-xxx	DirectPath	FADD/FMUL	
PSRLW mmreg1, mmreg2	0Fh	D1h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSRLW mmreg, mem64	0Fh	D1h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSRLW mmreg, imm8	0Fh	71h	11-010-xxx	DirectPath	FADD/FMUL	
PSUBB mmreg1, mmreg2	0Fh	F8h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBB mmreg, mem64	0Fh	F8h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSUBD mmreg1, mmreg2	0Fh	FAh	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBD mmreg, mem64	0Fh	FAh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSUBSB mmreg1, mmreg2	0Fh	E8h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBSB mmreg, mem64	0Fh	E8h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSUBSW mmreg1, mmreg2	0Fh	E9h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBSW mmreg, mem64	0Fh	E9h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSUBUSB mmreg1, mmreg2	0Fh	D8h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBUSB mmreg, mem64	0Fh	D8h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSUBUSW mmreg1, mmreg2	0Fh	D9h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBUSW mmreg, mem64	0Fh	D9h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PSUBW mmreg1, mmreg2	0Fh	F9h	11-xxx-xxx	DirectPath	FADD/FMUL	
PSUBW mmreg, mem64	0Fh	F9h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKHBW mmreg1, mmreg2	0Fh	68h	11-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKHBW mmreg, mem64	0Fh	68h	mm-xxx-xxx	DirectPath	FADD/FMUL	

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

Table 12. MMX™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Notes
PUNPCKHDQ mmreg1, mmreg2	0Fh	6Ah	11-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKHDQ mmreg, mem64	0Fh	6Ah	mm-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKHWD mmreg1, mmreg2	0Fh	69h	11-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKHWD mmreg, mem64	0Fh	69h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKLBW mmreg1, mmreg2	0Fh	60h	11-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKLBW mmreg, mem64	0Fh	60h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKLDQ mmreg1, mmreg2	0Fh	62h	11-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKLDQ mmreg, mem64	0Fh	62h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKLWD mmreg1, mmreg2	0Fh	61h	11-xxx-xxx	DirectPath	FADD/FMUL	
PUNPCKLWD mmreg, mem64	0Fh	61h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PXOR mmreg1, mmreg2	0Fh	EFh	11-xxx-xxx	DirectPath	FADD/FMUL	
PXOR mmreg, mem64	0Fh	EFh	mm-xxx-xxx	DirectPath	FADD/FMUL	

Notes:

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

Table 13. MMX™ Extensions

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Notes
MASKMOVQ mmreg1, mmreg2	0Fh	F7h		VectorPath	FADD/FMUL/FSTORE	
MOVNTQ mem64, mmreg	0Fh	E7h		DirectPath	FSTORE	
PAVGB mmreg1, mmreg2	0Fh	E0h	11-xxx-xxx	DirectPath	FADD/FMUL	
PAVGB mmreg, mem64	0Fh	E0h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PAVGW mmreg1, mmreg2	0Fh	E3h	11-xxx-xxx	DirectPath	FADD/FMUL	
PAVGW mmreg, mem64	0Fh	E3h	mm-xxx-xxx	DirectPath	FADD/FMUL	
PEXTRW reg32, mmreg, imm8	0Fh	C5h		VectorPath		
PINSRW mmreg, reg32, imm8	0Fh	C4h		VectorPath		
PINSRW mmreg, mem16, imm8	0Fh	C4h		VectorPath		
PMAXSW mmreg1, mmreg2	0Fh	EEh	11-xxx-xxx	DirectPath	FADD/FMUL	
PMAXSW mmreg, mem64	0Fh	EEh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PMAXUB mmreg1, mmreg2	0Fh	DEh	11-xxx-xxx	DirectPath	FADD/FMUL	
PMAXUB mmreg, mem64	0Fh	DEh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PMINSW mmreg1, mmreg2	0Fh	EAh	11-xxx-xxx	DirectPath	FADD/FMUL	

Notes:

1. For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.

Table 13. MMX™ Extensions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Notes
PMINSW mmreg, mem64	0Fh	EAh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PMINUB mmreg1, mmreg2	0Fh	DAh	11-xxx-xxx	DirectPath	FADD/FMUL	
PMINUB mmreg, mem64	0Fh	DAh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PMOVMSKB reg32, mmreg	0Fh	D7h		VectorPath		
PMULHUW mmreg1, mmreg2	0Fh	E4h	11-xxx-xxx	DirectPath	FMUL	
PMULHUW mmreg, mem64	0Fh	E4h	mm-xxx-xxx	DirectPath	FMUL	
PSADBW mmreg1, mmreg2	0Fh	F6h	11-xxx-xxx	DirectPath	FADD	
PSADBW mmreg, mem64	0Fh	F6h	mm-xxx-xxx	DirectPath	FADD	
PSHUFW mmreg1, mmreg2, imm8	0Fh	70h		DirectPath	FADD/FMUL	
PSHUFW mmreg, mem64, imm8	0Fh	70h		DirectPath	FADD/FMUL	
PREFETCHNTA mem8	0Fh	18h		DirectPath	-	1
PREFETCHT0 mem8	0Fh	18h		DirectPath	-	1
PREFETCHT1 mem8	0Fh	18h		DirectPath	-	1
PREFETCHT2 mem8	0Fh	18h		DirectPath	-	1
SFENCE	0Fh	AEh		VectorPath	-	

Notes:

1. For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.

Table 14. Floating-Point Instructions

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
F2XM1	D9h	F0h		VectorPath		
FABS	D9h	E1h		DirectPath	FMUL	
FADD ST, ST(i)	D8h		11-000-xxx	DirectPath	FADD	1
FADD [mem32real]	D8h		mm-000-xxx	DirectPath	FADD	
FADD ST(i), ST	DCh		11-000-xxx	DirectPath	FADD	1
FADD [mem64real]	DCh		mm-000-xxx	DirectPath	FADD	
FADDP ST(i), ST	DEh		11-000-xxx	DirectPath	FADD	1
FBLD [mem80]	DFh		mm-100-xxx	VectorPath		
FBSTP [mem80]	DFh		mm-110-xxx	VectorPath		
FCHS	D9h	E0h		DirectPath	FMUL	
FCLEX	DBh	E2h		VectorPath		

Notes:

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

Table 14. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
FCMOVB ST(0), ST(i)	DAh	C0-C7h		VectorPath		
FCMOVE ST(0), ST(i)	DAh	C8-CFh		VectorPath		
FCMOVBE ST(0), ST(i)	DAh	D0-D7h		VectorPath		
FCMOVU ST(0), ST(i)	DAh	D8-DFh		VectorPath		
FCMOVNB ST(0), ST(i)	DBh	C0-C7h		VectorPath		
FCMOVNE ST(0), ST(i)	DBh	C8-CFh		VectorPath		
FCMOVNBE ST(0), ST(i)	DBh	D0-D7h		VectorPath		
FCMOVNU ST(0), ST(i)	DBh	D8-DFh		VectorPath		
FCOM ST(i)	D8h		11-010-xxx	DirectPath	FADD	1
FCOMP ST(i)	D8h		11-011-xxx	DirectPath	FADD	1
FCOM [mem32real]	D8h		mm-010-xxx	DirectPath	FADD	
FCOM [mem64real]	DCh		mm-010-xxx	DirectPath	FADD	
FCOMI ST, ST(i)	DBh	F0-F7h		VectorPath	FADD	
FCOMIP ST, ST(i)	DFh	F0-F7h		VectorPath	FADD	
FCOMP [mem32real]	D8h		mm-011-xxx	DirectPath	FADD	
FCOMP [mem64real]	DCh		mm-011-xxx	DirectPath	FADD	
FCOMPP	DEh	D9h	11-011-001	DirectPath	FADD	
FCOS	D9h	FFh		VectorPath		
FDECSTP	D9h	F6h		DirectPath	FADD/FMUL/FSTORE	
FDIV ST, ST(i)	D8h		11-110-xxx	DirectPath	FMUL	1
FDIV ST(i), ST	DCh		11-111-xxx	DirectPath	FMUL	1
FDIV [mem32real]	D8h		mm-110-xxx	DirectPath	FMUL	
FDIV [mem64real]	DCh		mm-110-xxx	DirectPath	FMUL	
FDIVP ST, ST(i)	DEh		11-111-xxx	DirectPath	FMUL	1
FDIVR ST, ST(i)	D8h		11-110-xxx	DirectPath	FMUL	1
FDIVR ST(i), ST	DCh		11-111-xxx	DirectPath	FMUL	1
FDIVR [mem32real]	D8h		mm-111-xxx	DirectPath	FMUL	
FDIVR [mem64real]	DCh		mm-111-xxx	DirectPath	FMUL	
FDIVRP ST(i), ST	DEh		11-110-xxx	DirectPath	FMUL	1
FFREE ST(i)	DDh		11-000-xxx	DirectPath	FADD/FMUL/FSTORE	1
FFREEP ST(i)	DFh	C0-C7h		DirectPath	FADD/FMUL/FSTORE	1

Notes:

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

Table 14. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
FIADD [mem32int]	DAh		mm-000-xxx	VectorPath		
FIADD [mem16int]	DEh		mm-000-xxx	VectorPath		
FICOM [mem32int]	DAh		mm-010-xxx	VectorPath		
FICOM [mem16int]	DEh		mm-010-xxx	VectorPath		
FICOMP [mem32int]	DAh		mm-011-xxx	VectorPath		
FICOMP [mem16int]	DEh		mm-011-xxx	VectorPath		
FIDIV [mem32int]	DAh		mm-110-xxx	VectorPath		
FIDIV [mem16int]	DEh		mm-110-xxx	VectorPath		
FIDIVR [mem32int]	DAh		mm-111-xxx	VectorPath		
FIDIVR [mem16int]	DEh		mm-111-xxx	VectorPath		
FILD [mem16int]	DFh		mm-000-xxx	DirectPath	FSTORE	
FILD [mem32int]	DBh		mm-000-xxx	DirectPath	FSTORE	
FILD [mem64int]	DFh		mm-101-xxx	DirectPath	FSTORE	
FIMUL [mem32int]	DAh		mm-001-xxx	VectorPath		
FIMUL [mem16int]	DEh		mm-001-xxx	VectorPath		
FINCSTP	D9h	F7h		DirectPath	FADD/FMUL/FSTORE	
FINIT	DBh	E3h		VectorPath		
FIST [mem16int]	DFh		mm-010-xxx	DirectPath	FSTORE	
FIST [mem32int]	DBh		mm-010-xxx	DirectPath	FSTORE	
FISTP [mem16int]	DFh		mm-011-xxx	DirectPath	FSTORE	
FISTP [mem32int]	DBh		mm-011-xxx	DirectPath	FSTORE	
FISTP [mem64int]	DFh		mm-111-xxx	DirectPath	FSTORE	
FISUB [mem32int]	DAh		mm-100-xxx	VectorPath		
FISUB [mem16int]	DEh		mm-100-xxx	VectorPath		
FISUBR [mem32int]	DAh		mm-101-xxx	VectorPath		
FISUBR [mem16int]	DEh		mm-101-xxx	VectorPath		
FLD ST(i)	D9h		11-000-xxx	DirectPath	FADD/FMUL	1
FLD [mem32real]	D9h		mm-000-xxx	DirectPath	FADD/FMUL/FSTORE	
FLD [mem64real]	DDh		mm-000-xxx	DirectPath	FADD/FMUL/FSTORE	
FLD [mem80real]	DBh		mm-101-xxx	VectorPath		
FLD1	D9h	E8h		DirectPath	FSTORE	

Notes:

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

Table 14. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
FLDCW [mem16]	D9h		mm-101-xxx	VectorPath		
FLDENV [mem14byte]	D9h		mm-100-xxx	VectorPath		
FLDENV [mem28byte]	D9h		mm-100-xxx	VectorPath		
FLDL2E	D9h	EAh		DirectPath	FSTORE	
FLDL2T	D9h	E9h		DirectPath	FSTORE	
FLDLG2	D9h	ECh		DirectPath	FSTORE	
FLDLN2	D9h	EDh		DirectPath	FSTORE	
FLDPI	D9h	EBh		DirectPath	FSTORE	
FLDZ	D9h	EEh		DirectPath	FSTORE	
FMUL ST, ST(i)	D8h		11-001-xxx	DirectPath	FMUL	1
FMUL ST(i), ST	DCh		11-001-xxx	DirectPath	FMUL	1
FMUL [mem32real]	D8h		mm-001-xxx	DirectPath	FMUL	
FMUL [mem64real]	DCh		mm-001-xxx	DirectPath	FMUL	
FMULP ST, ST(i)	DEh		11-001-xxx	DirectPath	FMUL	1
FNOP	D9h	D0h		DirectPath	FADD/FMUL/FSTORE	
FPTAN	D9h	F2h		VectorPath		
FPATAN	D9h	F3h		VectorPath		
FPREM	D9h	F8h		DirectPath	FMUL	
FPREM1	D9h	F5h		DirectPath	FMUL	
FRNDINT	D9h	FCh		VectorPath		
FRSTOR [mem94byte]	DDh		mm-100-xxx	VectorPath		
FRSTOR [mem108byte]	DDh		mm-100-xxx	VectorPath		
FSAVE [mem94byte]	DDh		mm-110-xxx	VectorPath		
FSAVE [mem108byte]	DDh		mm-110-xxx	VectorPath		
FSCALE	D9h	FDh		VectorPath		
FSIN	D9h	FEh		VectorPath		
FSINCOS	D9h	FBh		VectorPath		
FSQRT	D9h	FAh		DirectPath	FMUL	
FST [mem32real]	D9h		mm-010-xxx	DirectPath	FSTORE	
FST [mem64real]	DDh		mm-010-xxx	DirectPath	FSTORE	
FST ST(i)	DDh		11-010xxx	DirectPath	FADD/FMUL	

Notes:

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

Table 14. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
FSTCW [mem16]	D9h		mm-111-xxx	VectorPath		
FSTENV [mem14byte]	D9h		mm-110-xxx	VectorPath		
FSTENV [mem28byte]	D9h		mm-110-xxx	VectorPath		
FSTP [mem32real]	D9h		mm-011-xxx	DirectPath	FADD/FMUL	
FSTP [mem64real]	DDh		mm-011-xxx	DirectPath	FADD/FMUL	
FSTP [mem80real]	D9h		mm-111-xxx	VectorPath		
FSTP ST(i)	DDh		11-011-xxx	DirectPath	FADD/FMUL	
FSTSW AX	DFh	E0h		VectorPath		
FSTSW [mem16]	DDh		mm-111-xxx	VectorPath	FSTORE	
FSUB [mem32real]	D8h		mm-100-xxx	DirectPath	FADD	
FSUB [mem64real]	DCh		mm-100-xxx	DirectPath	FADD	
FSUB ST, ST(i)	D8h		11-100-xxx	DirectPath	FADD	1
FSUB ST(i), ST	DCh		11-101-xxx	DirectPath	FADD	1
FSUBP ST, ST(i)	DEh		11-101-xxx	DirectPath	FADD	1
FSUBR [mem32real]	D8h		mm-101-xxx	DirectPath	FADD	
FSUBR [mem64real]	DCh		mm-101-xxx	DirectPath	FADD	
FSUBR ST, ST(i)	D8h		11-100-xxx	DirectPath	FADD	1
FSUBR ST(i), ST	DCh		11-101-xxx	DirectPath	FADD	1
FSUBRP ST(i), ST	DEh		11-100-xxx	DirectPath	FADD	1
FTST	D9h	E4h		DirectPath	FADD	
FUCOM	DDh		11-100-xxx	DirectPath	FADD	
FUCOMI ST, ST(i)	DB	E8-EFh		VectorPath	FADD	
FUCOMIP ST, ST(i)	DF	E8-EFh		VectorPath	FADD	
FUCOMP	DDh		11-101-xxx	DirectPath	FADD	
FUCOMPP	DAh	E9h		DirectPath	FADD	
FWAIT	9Bh			DirectPath		
FXAM	D9h	E5h		VectorPath		
FXCH	D9h		11-001-xxx	DirectPath	FADD/FMUL/FSTORE	
FXTRACT	D9h	F4h		VectorPath		
FYL2X	D9h	F1h		VectorPath		
FYL2XP1	D9h	F9h		VectorPath		

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

Table 15. 3DNow!™ Instructions

Instruction Mnemonic	Prefix Byte(s)	imm8	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
FEMMS	0Fh	0Eh		DirectPath	FADD/FMUL/FSTORE	2
PAVGUSB mmreg1, mmreg2	0Fh, 0Fh	BFh	11-xxx-xxx	DirectPath	FADD/FMUL	
PAVGUSB mmreg, mem64	0Fh, 0Fh	BFh	mm-xxx-xxx	DirectPath	FADD/FMUL	
PF2ID mmreg1, mmreg2	0Fh, 0Fh	1Dh	11-xxx-xxx	DirectPath	FADD	
PF2ID mmreg, mem64	0Fh, 0Fh	1Dh	mm-xxx-xxx	DirectPath	FADD	
PFACC mmreg1, mmreg2	0Fh, 0Fh	AEh	11-xxx-xxx	DirectPath	FADD	
PFACC mmreg, mem64	0Fh, 0Fh	AEh	mm-xxx-xxx	DirectPath	FADD	
PFADD mmreg1, mmreg2	0Fh, 0Fh	9Eh	11-xxx-xxx	DirectPath	FADD	
PFADD mmreg, mem64	0Fh, 0Fh	9Eh	mm-xxx-xxx	DirectPath	FADD	
PFCMPEQ mmreg1, mmreg2	0Fh, 0Fh	B0h	11-xxx-xxx	DirectPath	FADD	
PFCMPEQ mmreg, mem64	0Fh, 0Fh	B0h	mm-xxx-xxx	DirectPath	FADD	
PFCMPGE mmreg1, mmreg2	0Fh, 0Fh	90h	11-xxx-xxx	DirectPath	FADD	
PFCMPGE mmreg, mem64	0Fh, 0Fh	90h	mm-xxx-xxx	DirectPath	FADD	
PFCMPGT mmreg1, mmreg2	0Fh, 0Fh	A0h	11-xxx-xxx	DirectPath	FADD	
PFCMPGT mmreg, mem64	0Fh, 0Fh	A0h	mm-xxx-xxx	DirectPath	FADD	
PFMAX mmreg1, mmreg2	0Fh, 0Fh	A4h	11-xxx-xxx	DirectPath	FADD	
PFMAX mmreg, mem64	0Fh, 0Fh	A4h	mm-xxx-xxx	DirectPath	FADD	
PFMIN mmreg1, mmreg2	0Fh, 0Fh	94h	11-xxx-xxx	DirectPath	FADD	
PFMIN mmreg, mem64	0Fh, 0Fh	94h	mm-xxx-xxx	DirectPath	FADD	
PFMUL mmreg1, mmreg2	0Fh, 0Fh	B4h	11-xxx-xxx	DirectPath	FMUL	
PFMUL mmreg, mem64	0Fh, 0Fh	B4h	mm-xxx-xxx	DirectPath	FMUL	
PFRCP mmreg1, mmreg2	0Fh, 0Fh	96h	11-xxx-xxx	DirectPath	FMUL	
PFRCP mmreg, mem64	0Fh, 0Fh	96h	mm-xxx-xxx	DirectPath	FMUL	
PFRCPIT1 mmreg1, mmreg2	0Fh, 0Fh	A6h	11-xxx-xxx	DirectPath	FMUL	
PFRCPIT1 mmreg, mem64	0Fh, 0Fh	A6h	mm-xxx-xxx	DirectPath	FMUL	
PFRCPIT2 mmreg1, mmreg2	0Fh, 0Fh	B6h	11-xxx-xxx	DirectPath	FMUL	
PFRCPIT2 mmreg, mem64	0Fh, 0Fh	B6h	mm-xxx-xxx	DirectPath	FMUL	
PFRSQIT1 mmreg1, mmreg2	0Fh, 0Fh	A7h	11-xxx-xxx	DirectPath	FMUL	
PFRSQIT1 mmreg, mem64	0Fh, 0Fh	A7h	mm-xxx-xxx	DirectPath	FMUL	
PFRSQRT mmreg1, mmreg2	0Fh, 0Fh	97h	11-xxx-xxx	DirectPath	FMUL	

Notes:

1. *For the PREFETCH and PREFETCHW instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.*
2. *The byte listed in the column titled 'imm8' is actually the opcode byte.*

Table 15. 3DNow!™ Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	imm8	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
PFRSQRT mmreg, mem64	0Fh, 0Fh	97h	mm-xxx-xxx	DirectPath	FMUL	
PFSUB mmreg1, mmreg2	0Fh, 0Fh	9Ah	11-xxx-xxx	DirectPath	FADD	
PFSUB mmreg, mem64	0Fh, 0Fh	9Ah	mm-xxx-xxx	DirectPath	FADD	
PFSUBR mmreg1, mmreg2	0Fh, 0Fh	AAh	11-xxx-xxx	DirectPath	FADD	
PFSUBR mmreg, mem64	0Fh, 0Fh	AAh	mm-xxx-xxx	DirectPath	FADD	
PI2FD mmreg1, mmreg2	0Fh, 0Fh	0Dh	11-xxx-xxx	DirectPath	FADD	
PI2FD mmreg, mem64	0Fh, 0Fh	0Dh	mm-xxx-xxx	DirectPath	FADD	
PMULHRW mmreg1, mmreg2	0Fh, 0Fh	B7h	11-xxx-xxx	DirectPath	FMUL	
PMULHRW mmreg1, mem64	0Fh, 0Fh	B7h	mm-xxx-xxx	DirectPath	FMUL	
PREFETCH mem8	0Fh	0Dh	mm-000-xxx	DirectPath	-	1, 2
PREFETCHW mem8	0Fh	0Dh	mm-001-xxx	DirectPath	-	1, 2

Notes:

1. *For the PREFETCH and PREFETCHW instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.*
2. *The byte listed in the column titled 'imm8' is actually the opcode byte.*

Table 16. 3DNow!™ Extensions

Instruction Mnemonic	Prefix Byte(s)	imm8	ModR/M Byte	Decode Type	FPU Pipe(s)	Note
PF2IW mmreg1, mmreg2	0Fh, 0Fh	1Ch	11-xxx-xxx	DirectPath	FADD	
PF2IW mmreg, mem64	0Fh, 0Fh	1Ch	mm-xxx-xxx	DirectPath	FADD	
PFNACC mmreg1, mmreg2	0Fh, 0Fh	8Ah	11-xxx-xxx	DirectPath	FADD	
PFNACC mmreg, mem64	0Fh, 0Fh	8Ah	mm-xxx-xxx	DirectPath	FADD	
PFPNACC mmreg1, mmreg2	0Fh, 0Fh	8Eh	11-xxx-xxx	DirectPath	FADD	
PFPNACC mmreg, mem64	0Fh, 0Fh	8Eh	mm-xxx-xxx	DirectPath	FADD	
PI2FW mmreg1, mmreg2	0Fh, 0Fh	0Ch	11-xxx-xxx	DirectPath	FADD	
PI2FW mmreg, mem64	0Fh, 0Fh	0Ch	mm-xxx-xxx	DirectPath	FADD	
PSWAPD mmreg1, mmreg2	0Fh, 0Fh	BBh	11-xxx-xxx	DirectPath	FADD/FMUL	
PSWAPD mmreg, mem64	0Fh, 0Fh	BBh	mm-xxx-xxx	DirectPath	FADD/FMUL	

Appendix E

DirectPath versus VectorPath Instructions

Select DirectPath Over VectorPath Instructions

Use DirectPath instructions rather than VectorPath instructions. DirectPath instructions are optimized for decode and execute efficiently by minimizing the number of operations per x86 instruction, which includes 'register←register op memory' as well as 'register←register op register' forms of instructions.

DirectPath Instructions

The following tables contain DirectPath instructions, which should be used in the AMD Athlon processor wherever possible:

- Table 17, "DirectPath Integer Instructions," on page 176
- Table 18, "DirectPath MMX™ Instructions," on page 183 and Table 19, "DirectPath MMX™ Extensions," on page 184
- Table 20, "DirectPath Floating-Point Instructions," on page 185
- All 3DNow! instructions, including the 3DNow! Extensions, are DirectPath and are listed in Table 15, "3DNow!™ Instructions," on page 171 and Table 16, "3DNow!™ Extensions," on page 173.

Table 17. DirectPath Integer Instructions

Instruction Mnemonic
ADC mreg8, reg8
ADC mem8, reg8
ADC mreg16/32, reg16/32
ADC mem16/32, reg16/32
ADC reg8, mreg8
ADC reg8, mem8
ADC reg16/32, mreg16/32
ADC reg16/32, mem16/32
ADC AL, imm8
ADC EAX, imm16/32
ADC mreg8, imm8
ADC mem8, imm8
ADC mreg16/32, imm16/32
ADC mem16/32, imm16/32
ADC mreg16/32, imm8 (sign extended)
ADC mem16/32, imm8 (sign extended)
ADD mreg8, reg8
ADD mem8, reg8
ADD mreg16/32, reg16/32
ADD mem16/32, reg16/32
ADD reg8, mreg8
ADD reg8, mem8
ADD reg16/32, mreg16/32
ADD reg16/32, mem16/32
ADD AL, imm8
ADD EAX, imm16/32
ADD mreg8, imm8
ADD mem8, imm8
ADD mreg16/32, imm16/32
ADD mem16/32, imm16/32
ADD mreg16/32, imm8 (sign extended)
ADD mem16/32, imm8 (sign extended)
AND mreg8, reg8
AND mem8, reg8

Instruction Mnemonic
AND mreg16/32, reg16/32
AND mem16/32, reg16/32
AND reg8, mreg8
AND reg8, mem8
AND reg16/32, mreg16/32
AND reg16/32, mem16/32
AND AL, imm8
AND EAX, imm16/32
AND mreg8, imm8
AND mem8, imm8
AND mreg16/32, imm16/32
AND mem16/32, imm16/32
AND mreg16/32, imm8 (sign extended)
AND mem16/32, imm8 (sign extended)
BSWAP EAX
BSWAP ECX
BSWAP EDX
BSWAP EBX
BSWAP ESP
BSWAP EBP
BSWAP ESI
BSWAP EDI
BT mreg16/32, reg16/32
BT mreg16/32, imm8
BT mem16/32, imm8
CBW/CWDE
CLC
CMC
CMOVA/CMOVBE reg16/32, reg16/32
CMOVA/CMOVBE reg16/32, mem16/32
CMOVAE/CMOVNB/CMOVNC reg16/32, mem16/32
CMOVAE/CMOVNB/CMOVNC mem16/32, mem16/32
CMOVB/CMOVC/CMOVNAE reg16/32, reg16/32
CMOVB/CMOVC/CMOVNAE mem16/32, reg16/32

Table 17. DirectPath Integer Instructions (continued)

Instruction Mnemonic
CMOVBE/CMOVNA reg16/32, reg16/32
CMOVBE/CMOVNA reg16/32, mem16/32
CMOVE/CMOVZ reg16/32, reg16/32
CMOVE/CMOVZ reg16/32, mem16/32
CMOVG/CMOVNLE reg16/32, reg16/32
CMOVG/CMOVNLE reg16/32, mem16/32
CMOVGE/CMOVNL reg16/32, reg16/32
CMOVGE/CMOVNL reg16/32, mem16/32
CMOVL/CMOVNGE reg16/32, reg16/32
CMOVL/CMOVNGE reg16/32, mem16/32
CMOVLE/CMOVNG reg16/32, reg16/32
CMOVLE/CMOVNG reg16/32, mem16/32
CMOVNE/CMOVNZ reg16/32, reg16/32
CMOVNE/CMOVNZ reg16/32, mem16/32
CMOVNO reg16/32, reg16/32
CMOVNO reg16/32, mem16/32
CMOVNP/CMOVPO reg16/32, reg16/32
CMOVNP/CMOVPO reg16/32, mem16/32
CMOVNS reg16/32, reg16/32
CMOVNS reg16/32, mem16/32
CMOVO reg16/32, reg16/32
CMOVO reg16/32, mem16/32
CMOVP/CMOVPE reg16/32, reg16/32
CMOVP/CMOVPE reg16/32, mem16/32
CMOVS reg16/32, reg16/32
CMOVS reg16/32, mem16/32
CMP mreg8, reg8
CMP mem8, reg8
CMP mreg16/32, reg16/32
CMP mem16/32, reg16/32
CMP reg8, mreg8
CMP reg8, mem8
CMP reg16/32, mreg16/32
CMP reg16/32, mem16/32

Instruction Mnemonic
CMP AL, imm8
CMP EAX, imm16/32
CMP mreg8, imm8
CMP mem8, imm8
CMP mreg16/32, imm16/32
CMP mem16/32, imm16/32
CMP mreg16/32, imm8 (sign extended)
CMP mem16/32, imm8 (sign extended)
CWD/CDQ
DEC EAX
DEC ECX
DEC EDX
DEC EBX
DEC ESP
DEC EBP
DEC ESI
DEC EDI
DEC mreg8
DEC mem8
DEC mreg16/32
DEC mem16/32
INC EAX
INC ECX
INC EDX
INC EBX
INC ESP
INC EBP
INC ESI
INC EDI
INC mreg8
INC mem8
INC mreg16/32
INC mem16/32
JO short disp8

Table 17. DirectPath Integer Instructions (continued)

Instruction Mnemonic
JNO short disp8
JB/JNAE short disp8
JNB/JAE short disp8
JZ/JE short disp8
JNZ/JNE short disp8
JBE/JNA short disp8
JNBE/JA short disp8
JS short disp8
JNS short disp8
JP/JPE short disp8
JNP/JPO short disp8
JL/JNGE short disp8
JNL/JGE short disp8
JLE/JNG short disp8
JNLE/JG short disp8
JO near disp16/32
JNO near disp16/32
JB/JNAE near disp16/32
JNB/JAE near disp16/32
JZ/JE near disp16/32
JNZ/JNE near disp16/32
JBE/JNA near disp16/32
JNBE/JA near disp16/32
JS near disp16/32
JNS near disp16/32
JP/JPE near disp16/32
JNP/JPO near disp16/32
JL/JNGE near disp16/32
JNL/JGE near disp16/32
JLE/JNG near disp16/32
JNLE/JG near disp16/32
JMP near disp16/32 (direct)
JMP far disp32/48 (direct)
JMP disp8 (short)

Instruction Mnemonic
JMP near mreg16/32 (indirect)
JMP near mem16/32 (indirect)
LEA reg32, mem16/32
MOV mreg8, reg8
MOV mem8, reg8
MOV mreg16/32, reg16/32
MOV mem16/32, reg16/32
MOV reg8, mreg8
MOV reg8, mem8
MOV reg16/32, mreg16/32
MOV reg16/32, mem16/32
MOV AL, mem8
MOV EAX, mem16/32
MOV mem8, AL
MOV mem16/32, EAX
MOV AL, imm8
MOV CL, imm8
MOV DL, imm8
MOV BL, imm8
MOV AH, imm8
MOV CH, imm8
MOV DH, imm8
MOV BH, imm8
MOV EAX, imm16/32
MOV ECX, imm16/32
MOV EDX, imm16/32
MOV EBX, imm16/32
MOV ESP, imm16/32
MOV EBP, imm16/32
MOV ESI, imm16/32
MOV EDI, imm16/32
MOV mreg8, imm8
MOV mem8, imm8
MOV mreg16/32, imm16/32

Table 17. DirectPath Integer Instructions (continued)

Instruction Mnemonic
MOV mem16/32, imm16/32
MOVSX reg16/32, mreg8
MOVSX reg16/32, mem8
MOVSX reg32, mreg16
MOVSX reg32, mem16
MOVZX reg16/32, mreg8
MOVZX reg16/32, mem8
MOVZX reg32, mreg16
MOVZX reg32, mem16
NEG mreg8
NEG mem8
NEG mreg16/32
NEG mem16/32
NOP (XCHG EAX, EAX)
NOT mreg8
NOT mem8
NOT mreg16/32
NOT mem16/32
OR mreg8, reg8
OR mem8, reg8
OR mreg16/32, reg16/32
OR mem16/32, reg16/32
OR reg8, mreg8
OR reg8, mem8
OR reg16/32, mreg16/32
OR reg16/32, mem16/32
OR AL, imm8
OR EAX, imm16/32
OR mreg8, imm8
OR mem8, imm8
OR mreg16/32, imm16/32
OR mem16/32, imm16/32
OR mreg16/32, imm8 (sign extended)
OR mem16/32, imm8 (sign extended)

Instruction Mnemonic
PUSH EAX
PUSH ECX
PUSH EDX
PUSH EBX
PUSH ESP
PUSH EBP
PUSH ESI
PUSH EDI
PUSH imm8
PUSH imm16/32
RCL mreg8, imm8
RCL mreg16/32, imm8
RCL mreg8, 1
RCL mem8, 1
RCL mreg16/32, 1
RCL mem16/32, 1
RCL mreg8, CL
RCL mreg16/32, CL
RCR mreg8, imm8
RCR mreg16/32, imm8
RCR mreg8, 1
RCR mem8, 1
RCR mreg16/32, 1
RCR mem16/32, 1
RCR mreg8, CL
RCR mreg16/32, CL
ROL mreg8, imm8
ROL mem8, imm8
ROL mreg16/32, imm8
ROL mem16/32, imm8
ROL mreg8, 1
ROL mem8, 1
ROL mreg16/32, 1
ROL mem16/32, 1

Table 17. DirectPath Integer Instructions (continued)

Instruction Mnemonic
ROL mreg8, CL
ROL mem8, CL
ROL mreg16/32, CL
ROL mem16/32, CL
ROR mreg8, imm8
ROR mem8, imm8
ROR mreg16/32, imm8
ROR mem16/32, imm8
ROR mreg8, 1
ROR mem8, 1
ROR mreg16/32, 1
ROR mem16/32, 1
ROR mreg8, CL
ROR mem8, CL
ROR mreg16/32, CL
ROR mem16/32, CL
SAR mreg8, imm8
SAR mem8, imm8
SAR mreg16/32, imm8
SAR mem16/32, imm8
SAR mreg8, 1
SAR mem8, 1
SAR mreg16/32, 1
SAR mem16/32, 1
SAR mreg8, CL
SAR mem8, CL
SAR mreg16/32, CL
SAR mem16/32, CL
SBB mreg8, reg8
SBB mem8, reg8
SBB mreg16/32, reg16/32
SBB mem16/32, reg16/32
SBB reg8, mreg8
SBB reg8, mem8

Instruction Mnemonic
SBB reg16/32, mreg16/32
SBB reg16/32, mem16/32
SBB AL, imm8
SBB EAX, imm16/32
SBB mreg8, imm8
SBB mem8, imm8
SBB mreg16/32, imm16/32
SBB mem16/32, imm16/32
SBB mreg16/32, imm8 (sign extended)
SBB mem16/32, imm8 (sign extended)
SETO mreg8
SETO mem8
SETNO mreg8
SETNO mem8
SETB/SETC/SETNAE mreg8
SETB/SETC/SETNAE mem8
SETAE/SETNB/SETNC mreg8
SETAE/SETNB/SETNC mem8
SETE/SETZ mreg8
SETE/SETZ mem8
SETNE/SETNZ mreg8
SETNE/SETNZ mem8
SETBE/SETNA mreg8
SETBE/SETNA mem8
SETA/SETNBE mreg8
SETA/SETNBE mem8
SETS mreg8
SETS mem8
SETNS mreg8
SETNS mem8
SETP/SETPE mreg8
SETP/SETPE mem8
SETNP/SETPO mreg8
SETNP/SETPO mem8

Table 17. DirectPath Integer Instructions (continued)

Instruction Mnemonic
SETL/SETNGE mreg8
SETL/SETNGE mem8
SETGE/SETNL mreg8
SETGE/SETNL mem8
SETLE/SETNG mreg8
SETLE/SETNG mem8
SETG/SETNLE mreg8
SETG/SETNLE mem8
SHL/SAL mreg8, imm8
SHL/SAL mem8, imm8
SHL/SAL mreg16/32, imm8
SHL/SAL mem16/32, imm8
SHL/SAL mreg8, 1
SHL/SAL mem8, 1
SHL/SAL mreg16/32, 1
SHL/SAL mem16/32, 1
SHL/SAL mreg8, CL
SHL/SAL mem8, CL
SHL/SAL mreg16/32, CL
SHL/SAL mem16/32, CL
SHR mreg8, imm8
SHR mem8, imm8
SHR mreg16/32, imm8
SHR mem16/32, imm8
SHR mreg8, 1
SHR mem8, 1
SHR mreg16/32, 1
SHR mem16/32, 1
SHR mreg8, CL
SHR mem8, CL
SHR mreg16/32, CL
SHR mem16/32, CL
STC
SUB mreg8, reg8

Instruction Mnemonic
SUB mem8, reg8
SUB mreg16/32, reg16/32
SUB mem16/32, reg16/32
SUB reg8, mreg8
SUB reg8, mem8
SUB reg16/32, mreg16/32
SUB reg16/32, mem16/32
SUB AL, imm8
SUB EAX, imm16/32
SUB mreg8, imm8
SUB mem8, imm8
SUB mreg16/32, imm16/32
SUB mem16/32, imm16/32
SUB mreg16/32, imm8 (sign extended)
SUB mem16/32, imm8 (sign extended)
TEST mreg8, reg8
TEST mem8, reg8
TEST mreg16/32, reg16/32
TEST mem16/32, reg16/32
TEST AL, imm8
TEST EAX, imm16/32
TEST mreg8, imm8
TEST mem8, imm8
TEST mreg8, imm16/32
TEST mem8, imm16/32
WAIT
XCHG EAX, EAX
XOR mreg8, reg8
XOR mem8, reg8
XOR mreg16/32, reg16/32
XOR mem16/32, reg16/32
XOR reg8, mreg8
XOR reg8, mem8
XOR reg16/32, mreg16/32

Table 17. DirectPath Integer Instructions (continued)

Instruction Mnemonic
XOR reg16/32, mem16/32
XOR AL, imm8
XOR EAX, imm16/32
XOR mreg8, imm8
XOR mem8, imm8
XOR mreg16/32, imm16/32
XOR mem16/32, imm16/32
XOR mreg16/32, imm8 (sign extended)
XOR mem16/32, imm8 (sign extended)

Table 18. DirectPath MMX™ Instructions

Instruction Mnemonic
EMMS
MOVD mmreg, mem32
MOVD mem32, mmreg
MOVQ mmreg1, mmreg2
MOVQ mmreg, mem64
MOVQ mmreg2, mmreg1
MOVQ mem64, mmreg
PACKSSDW mmreg1, mmreg2
PACKSSDW mmreg, mem64
PACKSSWB mmreg1, mmreg2
PACKSSWB mmreg, mem64
PACKUSWB mmreg1, mmreg2
PACKUSWB mmreg, mem64
PADDB mmreg1, mmreg2
PADDB mmreg, mem64
PADDD mmreg1, mmreg2
PADDD mmreg, mem64
PADDSB mmreg1, mmreg2
PADDSB mmreg, mem64
PADDSW mmreg1, mmreg2
PADDSW mmreg, mem64
PADDUSB mmreg1, mmreg2
PADDUSB mmreg, mem64
PADDUSW mmreg1, mmreg2
PADDUSW mmreg, mem64
PADDW mmreg1, mmreg2
PADDW mmreg, mem64
PAND mmreg1, mmreg2
PAND mmreg, mem64
PANDN mmreg1, mmreg2
PANDN mmreg, mem64
PCMPEQB mmreg1, mmreg2
PCMPEQB mmreg, mem64
PCMPEQD mmreg1, mmreg2

Instruction Mnemonic
PCMPEQD mmreg, mem64
PCMPEQW mmreg1, mmreg2
PCMPEQW mmreg, mem64
PCMPGTB mmreg1, mmreg2
PCMPGTB mmreg, mem64
PCMPGTD mmreg1, mmreg2
PCMPGTD mmreg, mem64
PCMPGTW mmreg1, mmreg2
PCMPGTW mmreg, mem64
PMADDWD mmreg1, mmreg2
PMADDWD mmreg, mem64
PMULHW mmreg1, mmreg2
PMULHW mmreg, mem64
PMULLW mmreg1, mmreg2
PMULLW mmreg, mem64
POR mmreg1, mmreg2
POR mmreg, mem64
PSLLD mmreg1, mmreg2
PSLLD mmreg, mem64
PSLLD mmreg, imm8
PSLLQ mmreg1, mmreg2
PSLLQ mmreg, mem64
PSLLQ mmreg, imm8
PSLLW mmreg1, mmreg2
PSLLW mmreg, mem64
PSLLW mmreg, imm8
PSRAW mmreg1, mmreg2
PSRAW mmreg, mem64
PSRAW mmreg, imm8
PSRAD mmreg1, mmreg2
PSRAD mmreg, mem64
PSRAD mmreg, imm8
PSRLD mmreg1, mmreg2
PSRLD mmreg, mem64

Table 18. DirectPath MMX™ Instructions (continued)

Instruction Mnemonic
PSRLD mmreg, imm8
PSRLQ mmreg1, mmreg2
PSRLQ mmreg, mem64
PSRLQ mmreg, imm8
PSRLW mmreg1, mmreg2
PSRLW mmreg, mem64
PSRLW mmreg, imm8
PSUBB mmreg1, mmreg2
PSUBB mmreg, mem64
PSUBD mmreg1, mmreg2
PSUBD mmreg, mem64
PSUBSB mmreg1, mmreg2
PSUBSB mmreg, mem64
PSUBSW mmreg1, mmreg2
PSUBSW mmreg, mem64
PSUBUSB mmreg1, mmreg2
PSUBUSB mmreg, mem64
PSUBUSW mmreg1, mmreg2
PSUBUSW mmreg, mem64
PSUBW mmreg1, mmreg2
PSUBW mmreg, mem64
PUNPCKHBW mmreg1, mmreg2
PUNPCKHBW mmreg, mem64
PUNPCKHDQ mmreg1, mmreg2
PUNPCKHDQ mmreg, mem64
PUNPCKHWD mmreg1, mmreg2
PUNPCKHWD mmreg, mem64
PUNPCKLBW mmreg1, mmreg2
PUNPCKLBW mmreg, mem64
PUNPCKLDQ mmreg1, mmreg2
PUNPCKLDQ mmreg, mem64
PUNPCKLWD mmreg1, mmreg2
PUNPCKLWD mmreg, mem64
PXOR mmreg1, mmreg2
PXOR mmreg, mem64

Table 19. DirectPath MMX™ Extensions

Instruction Mnemonic
MOVNTQ mem64, mmreg
PAVGB mmreg1, mmreg2
PAVGB mmreg, mem64
PAVGW mmreg1, mmreg2
PAVGW mmreg, mem64
PMAXSW mmreg1, mmreg2
PMAXSW mmreg, mem64
PMAXUB mmreg1, mmreg2
PMAXUB mmreg, mem64
PMINSW mmreg1, mmreg2
PMINSW mmreg, mem64
PMINUB mmreg1, mmreg2
PMINUB mmreg, mem64
PMULHUW mmreg1, mmreg2
PMULHUW mmreg, mem64
PSADBW mmreg1, mmreg2
PSADBW mmreg, mem64
PSHUFW mmreg1, mmreg2, imm8
PSHUFW mmreg, mem64, imm8
PREFETCHNTA mem8
PREFETCHT0 mem8
PREFETCHT1 mem8
PREFETCHT2 mem8

Table 20. DirectPath Floating-Point Instructions

Instruction Mnemonic
FABS
FADD ST, ST(i)
FADD [mem32real]
FADD ST(i), ST
FADD [mem64real]
FADDP ST(i), ST
FCHS
FCOM ST(i)
FCOMP ST(i)
FCOM [mem32real]
FCOM [mem64real]
FCOMP [mem32real]
FCOMP [mem64real]
FCOMPP
FDECSTP
FDIV ST, ST(i)
FDIV ST(i), ST
FDIV [mem32real]
FDIV [mem64real]
FDIVP ST, ST(i)
FDIVR ST, ST(i)
FDIVR ST(i), ST
FDIVR [mem32real]
FDIVR [mem64real]
FDIVRP ST(i), ST
FFREE ST(i)
FFREEP ST(i)
FILD [mem16int]
FILD [mem32int]
FILD [mem64int]
FIMUL [mem32int]
FIMUL [mem16int]
FINCSTP
FIST [mem16int]

Instruction Mnemonic
FIST [mem32int]
FISTP [mem16int]
FISTP [mem32int]
FISTP [mem64int]
FLD ST(i)
FLD [mem32real]
FLD [mem64real]
FLD [mem80real]
FLD1
FLDL2E
FLDL2T
FLDLG2
FLDLN2
FLDPI
FLDZ
FMUL ST, ST(i)
FMUL ST(i), ST
FMUL [mem32real]
FMUL [mem64real]
FMULP ST, ST(i)
FNOP
FPREM
FPREM1
FSQRT
FST [mem32real]
FST [mem64real]
FST ST(i)
FSTP [mem32real]
FSTP [mem64real]
FSTP [mem80real]
FSTP ST(i)
FSUB [mem32real]
FSUB [mem64real]
FSUB ST, ST(i)

Table 20. DirectPath Floating-Point Instructions

Instruction Mnemonic
FSUB ST(i), ST
FSUBP ST, ST(i)
FSUBR [mem32real]
FSUBR [mem64real]
FSUBR ST, ST(i)
FSUBR ST(i), ST
FSUBRP ST(i), ST
FTST
FUCOM
FUCOMP
FUCOMPP
FWAIT
FXCH

VectorPath Instructions

The following tables contain **VectorPath** instructions, which should be **avoided** in the AMD Athlon processor:

- Table 21, "VectorPath Integer Instructions," on page 187
- Table 22, "VectorPath MMX™ Instructions," on page 190 and Table 23, "VectorPath MMX™ Extensions," on page 190
- Table 24, "VectorPath Floating-Point Instructions," on page 191

Table 21. VectorPath Integer Instructions

Instruction Mnemonic
AAA
AAD
AAM
AAS
ARPL mreg16, reg16
ARPL mem16, reg16
BOUND
BSF reg16/32, mreg16/32
BSF reg16/32, mem16/32
BSR reg16/32, mreg16/32
BSR reg16/32, mem16/32
BT mem16/32, reg16/32
BTC mreg16/32, reg16/32
BTC mem16/32, reg16/32
BTC mreg16/32, imm8
BTC mem16/32, imm8
BTR mreg16/32, reg16/32
BTR mem16/32, reg16/32
BTR mreg16/32, imm8
BTR mem16/32, imm8
BTS mreg16/32, reg16/32
BTS mem16/32, reg16/32
BTS mreg16/32, imm8

Instruction Mnemonic
BTS mem16/32, imm8
CALL full pointer
CALL near imm16/32
CALL mem16:16/32
CALL near mreg32 (indirect)
CALL near mem32 (indirect)
CLD
CLI
CLTS
CMPSB mem8,mem8
CMPSW mem16, mem32
CMPSD mem32, mem32
CMPXCHG mreg8, reg8
CMPXCHG mem8, reg8
CMPXCHG mreg16/32, reg16/32
CMPXCHG mem16/32, reg16/32
CMPXCHG8B mem64
CPUID
DAA
DAS
DIV AL, mreg8
DIV AL, mem8
DIV EAX, mreg16/32
DIV EAX, mem16/32

Table 21. VectorPath Integer Instructions (continued)

Instruction Mnemonic
ENTER
IDIV mreg8
IDIV mem8
IDIV EAX, mreg16/32
IDIV EAX, mem16/32
IMUL reg16/32, imm16/32
IMUL reg16/32, mreg16/32, imm16/32
IMUL reg16/32, mem16/32, imm16/32
IMUL reg16/32, imm8 (sign extended)
IMUL reg16/32, mreg16/32, imm8 (signed)
IMUL reg16/32, mem16/32, imm8 (signed)
IMUL AX, AL, mreg8
IMUL AX, AL, mem8
IMUL EDX:EAX, EAX, mreg16/32
IMUL EDX:EAX, EAX, mem16/32
IMUL reg16/32, mreg16/32
IMUL reg16/32, mem16/32
IN AL, imm8
IN AX, imm8
IN EAX, imm8
IN AL, DX
IN AX, DX
IN EAX, DX
INVD
INVLPG
JCXZ/JEC short disp8
JMP far disp32/48 (direct)
JMP far mem32 (indirect)
JMP far mreg32 (indirect)
LAHF
LAR reg16/32, mreg16/32
LAR reg16/32, mem16/32
LDS reg16/32, mem32/48
LEA reg16, mem16/32

Instruction Mnemonic
LEAVE
LES reg16/32, mem32/48
LFS reg16/32, mem32/48
LGDT mem48
LGS reg16/32, mem32/48
LIDT mem48
LLDT mreg16
LLDT mem16
LMSW mreg16
LMSW mem16
LODSB AL, mem8
LODSW AX, mem16
LODSD EAX, mem32
LOOP disp8
LOOPE/LOOPZ disp8
LOOPNE/LOOPNZ disp8
LSL reg16/32, mreg16/32
LSL reg16/32, mem16/32
LSS reg16/32, mem32/48
LTR mreg16
LTR mem16
MOV mreg16, segment reg
MOV mem16, segment reg
MOV segment reg, mreg16
MOV segment reg, mem16
MOVSB mem8,mem8
MOVSD mem16, mem16
MOVSW mem32, mem32
MUL AL, mreg8
MUL AL, mem8
MUL AX, mreg16
MUL AX, mem16
MUL EAX, mreg32
MUL EAX, mem32

Table 21. VectorPath Integer Instructions (continued)

Instruction Mnemonic
OUT imm8, AL
OUT imm8, AX
OUT imm8, EAX
OUT DX, AL
OUT DX, AX
OUT DX, EAX
POP ES
POP SS
POP DS
POP FS
POP GS
POP EAX
POP ECX
POP EDX
POP EBX
POP ESP
POP EBP
POP ESI
POP EDI
POP mreg 16/32
POP mem 16/32
POPA/POPAD
POPF/POPFD
PUSH ES
PUSH CS
PUSH FS
PUSH GS
PUSH SS
PUSH DS
PUSH mreg16/32
PUSH mem16/32
PUSHA/PUSHAD
PUSHF/PUSHFD
RCL mem8, imm8

Instruction Mnemonic
RCL mem16/32, imm8
RCL mem8, CL
RCL mem16/32, CL
RCR mem8, imm8
RCR mem16/32, imm8
RCR mem8, CL
RCR mem16/32, CL
RDMSR
RDPMC
RDTSC
RET near imm16
RET near
RET far imm16
RET far
SAHF
SCASB AL, mem8
SCASW AX, mem16
SCASD EAX, mem32
SGDT mem48
SIDT mem48
SHLD mreg16/32, reg16/32, imm8
SHLD mem16/32, reg16/32, imm8
SHLD mreg16/32, reg16/32, CL
SHLD mem16/32, reg16/32, CL
SHRD mreg16/32, reg16/32, imm8
SHRD mem16/32, reg16/32, imm8
SHRD mreg16/32, reg16/32, CL
SHRD mem16/32, reg16/32, CL
SLDT mreg16
SLDT mem16
SMSW mreg16
SMSW mem16
STD
STI

Table 21. VectorPath Integer Instructions (continued)

Instruction Mnemonic
STOSB mem8, AL
STOSW mem16, AX
STOSD mem32, EAX
STR mreg16
STR mem16
SYSCALL
SYSENTER
SYSEXIT
SYSRET
VERR mreg16
VERR mem16
VERW mreg16
VERW mem16
WBINVD
WRMSR
XADD mreg8, reg8
XADD mem8, reg8
XADD mreg16/32, reg16/32
XADD mem16/32, reg16/32
XCHG reg8, mreg8
XCHG reg8, mem8
XCHG reg16/32, mreg16/32
XCHG reg16/32, mem16/32
XCHG EAX, ECX
XCHG EAX, EDX
XCHG EAX, EBX
XCHG EAX, ESP
XCHG EAX, EBP
XCHG EAX, ESI
XCHG EAX, EDI
XLAT

Table 22. VectorPath MMX™ Instructions

Instruction Mnemonic
MOVD mmreg, mreg32
MOVD mreg32, mmreg

Table 23. VectorPath MMX™ Extensions

Instruction Mnemonic
MASKMOVQ mmreg1, mmreg2
PEXTRW reg32, mmreg, imm8
PINSRW mmreg, reg32, imm8
PINSRW mmreg, mem16, imm8
PMOVMSKB reg32, mmreg
SFENCE

Table 24. VectorPath Floating-Point Instructions

Instruction Mnemonic
F2XM1
FBLD [mem80]
FBSTP [mem80]
FCLEX
FCMOVB ST(0), ST(i)
FCMOVE ST(0), ST(i)
FCMOVBE ST(0), ST(i)
FCMOVU ST(0), ST(i)
FCMOVNB ST(0), ST(i)
FCMOVNE ST(0), ST(i)
FCMOVNBE ST(0), ST(i)
FCMOVNU ST(0), ST(i)
FCOMI ST, ST(i)
FCOMIP ST, ST(i)
FCOS
FIADD [mem32int]
FIADD [mem16int]
FICOM [mem32int]
FICOM [mem16int]
FICOMP [mem32int]
FICOMP [mem16int]
FIDIV [mem32int]
FIDIV [mem16int]
FIDIVR [mem32int]
FIDIVR [mem16int]
FIMUL [mem32int]
FIMUL [mem16int]
FINIT
FISUB [mem32int]
FISUB [mem16int]
FISUBR [mem32int]
FISUBR [mem16int]
FLD [mem80real]
FLDCW [mem16]

Instruction Mnemonic
FLDENV [mem14byte]
FLDENV [mem28byte]
FPTAN
FPATAN
FRNDINT
FRSTOR [mem94byte]
FRSTOR [mem108byte]
FSAVE [mem94byte]
FSAVE [mem108byte]
FSCALE
FSIN
FSINCOS
FSTCW [mem16]
FSTENV [mem14byte]
FSTENV [mem28byte]
FSTP [mem80real]
FSTSW AX
FSTSW [mem16]
FUCOMI ST, ST(i)
FUCOMIP ST, ST(i)
FXAM
FXTRACT
FYL2X
FYL2XP1

Appendix F

Performance Monitoring Counters

This chapter describes how to use the AMD Athlon™ processor performance monitoring counters.

Overview

The AMD Athlon processor provides four 48-bit performance counters, which allows four types of events to be monitored simultaneously. These counters can either count events or measure duration. When counting events, a counter is incremented each time a specified event takes place or a specified number of events takes place. When measuring duration, a counter counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level. Table 26 on page 196 lists the events that can be counted with the performance monitoring counters.

Performance Counter Usage

The performance monitoring counters are supported by eight MSRs—PerfEvtSel[3:0] are the performance event select MSRs, and PerfCtr[3:0] are the performance counter MSRs.

These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively.

The PerfEvtSel[3:0] registers are located at MSR locations C001_0000h to C001_0003h. The PerfCtr[3:0] registers are located at MSR locations C001_0004h to C0001_0007h and are 64-byte registers.

The PerfEvtSel[3:0] registers can be accessed using the RDMSR/WRMSR instructions only when operating at privilege level 0. The PerfCtr[3:0] MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction, if the PCE flag in CR4 is set.

PerfEvtSel[3:0] MSRs (MSR Addresses C001_0000h–C001_0003h)

The PerfEvtSel[3:0] MSRs, shown in Figure 11, control the operation of the performance-monitoring counters, with one register used to set up each counter. These MSRs specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. The functions of the flags and fields within these MSRs are as are described in the following sections.

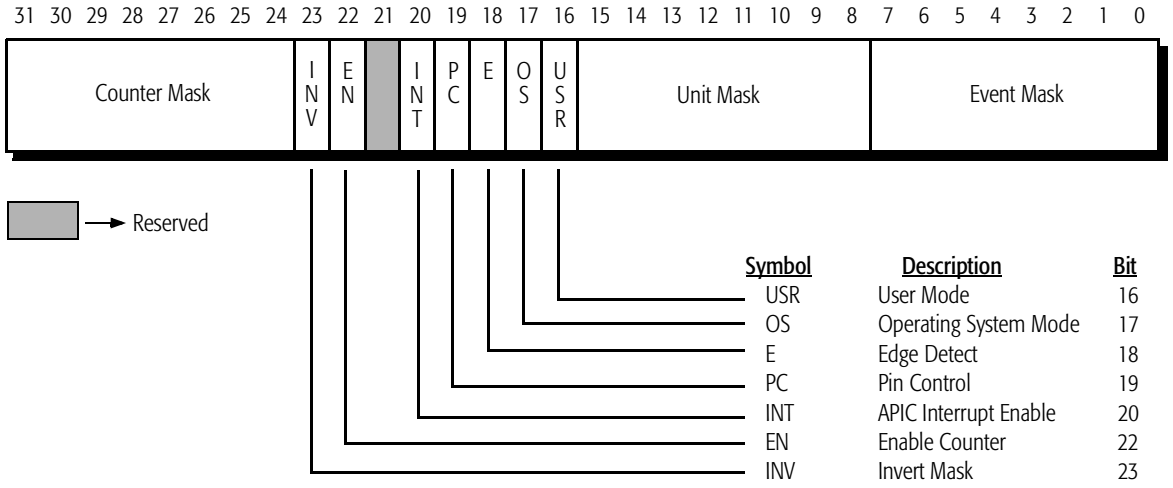

Figure 11. PerfEvtSel[3:0] Registers

Event Select Field (Bits 0–7) These bits are used to select the event to be monitored. See Table 26 on page 196 for a list of event masks and their 8-bit codes.

Unit Mask Field (Bits 8–15) These bits are used to further qualify the event selected in the event select field. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states. See Table 26 on page 196 for a list of unit masks and their 8-bit codes.

USR (User Mode) Flag (Bit 16) Events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.

OS (Operating System Mode) Flag (Bit 17) Events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.

E (Edge Detect) Flag (Bit 18) When this flag is set, edge detection of events is enabled. The processor counts the number of negated-to-asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

PC (Pin Control) Flag (Bit 19) When this flag is set, the processor toggles the PMi pins when the counter overflows. When this flag is clear, the processor toggles the PMi pins and increments the counter when performance monitoring events occur. The toggling of a pin is defined as assertion of the pin for one bus clock followed by negation.

INT (APIC Interrupt Enable) Flag (Bit 20) When this flag is set, the processor generates an interrupt through its local APIC on counter overflow.

EN (Enable Counter) Flag (Bit 22) This flag enables/disables the PerfEvtSeln MSR. When set, performance counting is enabled for this counter. When clear, this counter is disabled.

INV (Invert) Flag (Bit 23) By inverting the Counter Mask Field, this flag inverts the result of the counter comparison, allowing both greater than and less than comparisons.

Counter Mask Field (Bits 31–24) For events which can have multiple occurrences within one clock, this field is used to set a threshold. If the field is non-zero, the counter increments each time the number of events is

greater than or equal to the counter mask. Otherwise if this field is zero, then the counter increments by the total number of events.

Table 26. Performance Monitoring Counters

Event Number	Source Unit	Notes / Unit Mask (bits 15–8)	Event Description
20h	LS	1xxx_xxxxb = reserved x1xx_xxxxb = HS xx1x_xxxxb = GS xxx1_xxxxb = FS xxxx_1xxxb = DS xxxx_x1xxb = SS xxxx_xx1xb = CS xxxx_xxx1b = ES	Segment register loads
21h	LS		Stores to active instruction stream
40h	DC		Data cache accesses
41h	DC		Data cache misses
42h	DC	xxx1_xxxxb = Modified (M) xxxx_1xxxb = Owner (O) xxxx_x1xxb = Exclusive (E) xxxx_xx1xb = Shared (S) xxxx_xxx1b = Invalid (I)	Data cache refills
43h	DC	xxx1_xxxxb = Modified (M) xxxx_1xxxb = Owner (O) xxxx_x1xxb = Exclusive (E) xxxx_xx1xb = Shared (S) xxxx_xxx1b = Invalid (I)	Data cache refills from system
44h	DC	xxx1_xxxxb = Modified (M) xxxx_1xxxb = Owner (O) xxxx_x1xxb = Exclusive (E) xxxx_xx1xb = Shared (S) xxxx_xxx1b = Invalid (I)	Data cache writebacks
45h	DC		L1 DTLB misses and L2 DTLB hits
46h	DC		L1 and L2 DTLB misses
47h	DC		Misaligned data references
64h	BU		DRAM system requests

Table 26. Performance Monitoring Counters (continued)

Event Number	Source Unit	Notes / Unit Mask (bits 15–8)	Event Description
65h	BU	1xxx_xxxxb = reserved x1xx_xxxxb = WB xx1x_xxxxb = WP xxx1_xxxxb = WT bits 11-10 = reserved xxxx_xx1xb = WC xxxx_xxx1b = UC	System requests with the selected type
73h	BU	bits 15-11 = reserved xxxx_x1xxb = L2 (L2 hit and no DC hit) xxxx_xx1xb = Data cache xxxx_xxx1b = Instruction cache	Snoop hits
74h	BU	bits 15-10 = reserved xxxx_xx1xb = L2 single bit error xxxx_xxx1b = System single bit error	Single-bit ECC errors detected/corrected
75h	BU	bits 15-12 = reserved xxxx_1xxxb = I invalidates D xxxx_x1xxb = I invalidates I xxxx_xx1xb = D invalidates D xxxx_xxx1b = D invalidates I	Internal cache line invalidates
76h	BU		Cycles processor is running (not in HLT or STPCLK)
79h	BU	1xxx_xxxxb = Data block write from the L2 (TLB RMW) x1xx_xxxxb = Data block write from the DC xx1x_xxxxb = Data block write from the system xxx1_xxxxb = Data block read data store xxxx_1xxxb = Data block read data load xxxx_x1xxb = Data block read instruction xxxx_xx1xb = Tag write xxxx_xxx1b = Tag read	L2 requests

Table 26. Performance Monitoring Counters (continued)

Event Number	Source Unit	Notes / Unit Mask (bits 15–8)	Event Description
7Ah	BU		Cycles that at least one fill request waited to use the L2
80h	PC		Instruction cache fetches
81h	PC		Instruction cache misses
82h	PC		Instruction cache refills from L2
83h	PC		Instruction cache refills from system
84h	PC		L1 ITLB misses (and L2 ITLB hits)
85h	PC		(L1 and) L2 ITLB misses
86h	PC		Snoop resyncs
87h	PC		Instruction fetch stall cycles
88h	PC		Return stack hits
89h	PC		Return stack overflow
C0h	FR		Retired instructions (includes exceptions, interrupts, resyncs)
C1h	FR		Retired Ops
C2h	FR		Retired branches (conditional, unconditional, exceptions, interrupts)
C3h	FR		Retired branches mispredicted
C4h	FR		Retired taken branches
C5h	FR		Retired taken branches mispredicted
C6h	FR		Retired far control transfers
C8h	FR		Retired near returns
C9h	FR		Retired near returns mispredicted
CAh	FR		Retired indirect branches with target mispredicted
CDh	FR		Interrupts masked cycles (IF=0)
CEh	FR		Interrupts masked while pending cycles (INTR while IF=0)
CFh	FR		Number of taken hardware interrupts
D0h	FR		Instruction decoder empty
D1h	FR		Dispatch stalls (event masks D2h through DAh below combined)
D2h	FR		Branch abort to retire
D3h	FR		Serialize
D4h	FR		Segment load stall

Table 26. Performance Monitoring Counters (continued)

Event Number	Source Unit	Notes / Unit Mask (bits 15–8)	Event Description
D5h	FR		ICU full
D6h	FR		Reservation stations full
D7h	FR		FPU full
D8h	FR		LS full
D9h	FR		All quiet stall
DAh	FR		Far transfer or resync branch pending
DCh	FR		Breakpoint matches for DR0
DDh	FR		Breakpoint matches for DR1
DEh	FR		Breakpoint matches for DR2
DFh	FR		Breakpoint matches for DR3

PerfCtr[3:0] MSRs (MSR Addresses C001_0004h–C001_0007h)

The performance-counter MSRs contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Therefore, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions can begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization, which disables direct user access to the performance-monitoring counters but provides a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr[3:0]). Instead, the value should be treated as 64-bit sign extended, which

allows writing both positive and negative values to the performance counters. The performance counters may be initialized using a 64-bit signed integer in the range -2^{47} and $+2^{47}$. Negative values are useful for generating an interrupt after a specific number of events.

Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in one or more of the PerfEvtSel[3:0] MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction, which sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel[3:0] MSRs.

Event and Time-Stamp Monitoring Software

For applications to use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking
- Initialize and start counters
- Stop counters
- Read the event counters
- Reading of the time stamp counter

The event monitor feature determination procedure must determine whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. In addition, the procedure checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialization and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be

counted and the method used to count them and initializes the counter MSRs (PerfCtr[3:0]) to starting counts. The stop counters procedure stops the performance counters. (See "Starting and Stopping the Performance-Monitoring Counters" on page 200 for more information about starting and stopping the counters.)

The read counters procedure reads the values in the PerfCtr[3:0] MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures can be used instead of enabling the RDTSC and RDPMC instructions, which allow application code to read the counters directly.

Monitoring Counter Overflow

The AMD Athlon processor provides the option of generating a debug interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in one of the PerfEvtSel[3:0] MSRs. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following:

- Provide an interrupt routine for handling the counter overflow as an APIC interrupt
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt
- Reset the counter to its initial setting and return from the interrupt

An event monitor application utility or another application program can read the collected performance information of the profiled application.