Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors

Publication # **25112** Revision: **3.03**
Issue Date: **September 2003**

Trademarks

Contents

Figures

Tables

Revision History

Date	Rev.	Description
September 2003	3.03	Made several minor typographical and formatting corrections.
July 2003	3.02	Added index references. Corrected information pertaining to L1 and L2 data and instruction caches. Corrected information on alignment in Chapter 5, “Cache and Memory Optimizations”. Amended latency information in Appendix C.
April 2003	3.01	Clarified section 2.22 'Using Signed Integers for 32-Bit Array Indices'. Corrected factual errors and removed misleading examples from Cache and Memory chapter..
April 2003	3.00	Initial public release.

Chapter 1 Introduction

This guide provides optimization information and recommendations for the AMD Athlon™ 64 and AMD Opteron™ processors. These optimizations are designed to yield software code that is fast, compact, and efficient. Toward this end, the optimizations in each of the following chapters are listed in order of importance.

This chapter covers the following topics:

1.1 Intended Audience

This book is intended for compiler and assembler designers, as well as C, C++, and assembly-language programmers writing performance-sensitive code sequences. This guide assumes that you are familiar with the AMD64 instruction set and the AMD64 architecture (registers and programming modes). For complete information on the AMD64 architecture and instruction set, see the multivolume *AMD64 Architecture Programmer's Manual* available from AMD.com. Documentation volumes and their order numbers are provided below.

Title	Order no.
Volume 1, *Application Programming*	24592
Volume 2, *System Programming*	24593
Volume 3, *General-Purpose and System Instructions*	24594
Volume 4, *128-Bit Media Instructions*	26568
Volume 5, *64-Bit Media and x87 Floating-Point Instructions*	26569

1.2 Getting Started Quickly

More experienced readers may skip to "Key Optimizations" on page 6, which identifies the most important optimizations.

1.3 Using This Guide

This chapter explains how to get the most benefit from this guide. It defines important new terms you'll need to understand before reading the rest of this guide and lists the most important optimizations by rank.

Chapter 2 describes techniques that you can use to optimize your C and C++ source code. The "Application" section for each optimization indicates whether the optimization applies to 32-bit software, 64-bit software, or both.

Chapter 3 presents general assembly-language optimizations that improve the performance of software designed to run in 64-bit mode. All optimizations in this chapter apply only to 64-bit software.

The remaining chapters describe assembly-language optimizations. The "Application" section under each optimization indicates whether the optimization applies to 32-bit software, 64-bit software, or both.

Chapter 4	Instruction-Decoding Optimizations
Chapter 5	Cache and Memory Optimizations
Chapter 6	Branch Optimizations
Chapter 7	Scheduling Optimizations
Chapter 8	Integer Optimizations
Chapter 9	Optimizing with SIMD Instructions
Chapter 10	x87 Floating-Point Optimizations

Appendix A discusses the internal design, or microarchitecture, of the processor and provides specifications on the translation-lookaside buffers. It also provides information on other functional units that are not part of the main processor but are integrated on the chip.

Appendix B describes the memory write-combining feature of the processor.

Appendix CA provides a complete listing of all AMD64 instructions. It shows each instruction's encoding, decode type, execution latency, and—where applicable—the pipe used in the floating-point unit.

Appendix D discusses optimizations that improve the throughput of AGP transfers.

Appendix E describes coding practices that improve performance when using SSE and SSE2 instructions.

Special Information

Special information in this guide looks like this:

❖This symbol appears next to the most important, or *key*, optimizations.

Numbering Systems

The following suffixes identify different numbering systems:

This suffix	Identifies a
b	Binary number. For example, the binary equivalent of the number 5 is written 101b.
d	Decimal number. Decimal numbers are followed by this suffix only when the possibility of confusion exists. In general, decimal numbers are shown without a suffix.
h	Hexadecimal number. For example, the hexadecimal equivalent of the number 60 is written 3Ch.

Typographic Notation

This guide uses the following typographic notations for certain types of information:

This type of text	Identifies
italic	Placeholders that represent information you must provide. Italicized text is also used for the titles of publications and for emphasis.
`monowidth`	Program statements and function names.

Providing Feedback

If you have suggestions for improving this guide, we would like to hear from you. Please send your comments to the following e-mail address:

code.optimization@amd.com

1.4 Important New Terms

This section defines several important terms and concepts used in this guide.

Primitive Operations

AMD Athlon 64 and AMD Opteron processors perform four types of *primitive operations*:

- Integer (arithmetic or logic)
- Floating-point (arithmetic)
- Load
- Store

Internal Instruction Formats

The AMD64 instruction set is complex; instructions have variable-length encodings and many perform multiple primitive operations. AMD Athlon 64 and AMD Opteron processors do not execute these complex instructions directly, but, instead, decode them internally into simpler fixed-length instructions called *macro-ops*. Processor schedulers subsequently break down macro-ops into sequences of even simpler instructions called *micro-ops*, each of which specifies a single primitive operation.

A *macro-op* is a fixed-length instruction that:

- Expresses, at most, one integer or floating-point operation and one load and/or store operation.
- Is the primary unit of work managed (that is, dispatched and retired) by the processor.

A *micro-op* is a fixed-length instruction that:

- Expresses one and only one of the primitive operations that the processor can perform (for example, a load).
- Is executed by the processor's execution units.

Table 1 summarizes the differences between AMD64 instructions, macro-ops, and micro-ops.

Table 1. Instructions, Macro-ops and Micro-ops

Comparing	AMD64 instructions	Macro-ops	Micro-ops
Complexity	Complex A single instruction may specify one or more of each of the following operations: • Integer or floating-point operation • Load • Store	Average A single macro-op may specify—at most—one integer or floating-point operation and one of the following operations: • Load • Store • Load and store to the same address	Simple A single micro-op specifies only one of the following primitive operations: • Integer or floating-point • Load • Store
Encoded length	Variable (instructions are different lengths)	Fixed (all macro-ops are the same length)	Fixed (all micro-ops are the same length)
Regularized instruction fields	No (field locations and definitions vary among instructions)	Yes (field locations and definitions are the same for all macro-ops)	Yes (field locations and definitions are the same for all micro-ops)

Types of Instructions

Instructions are classified according to how they are decoded by the processor. There are three types of instructions:

Instruction Type	Description
DirectPath Single	A relatively common instruction that the processor decodes directly into one macro-op in hardware.
DirectPath Double	A relatively common instruction that the processor decodes directly into two macro-ops in hardware.
VectorPath	A sophisticated or less common instruction that the processor decodes into one or more (usually three or more) macro-ops using the on-chip microcode-engine ROM (MROM).

1.5 Key Optimizations

While all of the optimizations in this guide help improve software performance, some of them have more impact than others. Optimizations that offer the most improvement are called *key* optimizations.

Guideline

Concentrate your efforts on implementing key optimizations before moving on to other optimizations, and incorporate higher-ranking key optimizations first.

Key Optimizations by Rank

Table 1 lists the key optimizations by rank.

Table 2. Optimizations by Rank

Rank	Optimization	Page
1	Memory-Size Mismatches	88
2	Natural Alignment of Data Objects	91
3	Appropriate Memory Copying Routines	112
4	Density of Branches	126
5	Prefetch Instructions	97
6	Two-Byte Near-Return RET Instruction	128
7	DirectPath Instructions	70
8	Load-Execute Integer Instructions	71
9	Load-Execute Floating-Point Instructions with Floating-Point Operands	72
10	Load-Execute Floating-Point Instructions with Integer Operands	72
11	Write-combining	105
12	Branches That Depend on Random Data	130
13	SSE and SSE2 Instruction and Data Types	351
14	Placing Code and Data in the Same 64-Byte Cache Line	108

Chapter 2 C and C++ Source-Level Optimizations

Although C and C++ compilers generally produce very compact object code, many performance improvements are possible by careful source code optimization. Most such optimizations result from taking advantage of the underlying mechanisms used by C and C++ compilers to translate source code into sequences of AMD64 instructions. This chapter includes guidelines for writing C and C++ source code that result in the most efficiently optimized AMD64 code.

This chapter covers the following topics:

2.1 Declarations of Floating-Point Values

Optimization

When working with single precision (`float`) values:

- Use the `f` or `F` suffix (for example, `3.14f`) to specify a constant value of type `float`.
- Use function prototypes for all functions that accept arguments of type `float`.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

C and C++ compilers treat floating-point constants and arguments as double precision (`double`) unless you specify otherwise. However, single precision floating-point values occupy half the memory space as double precision values and can often provide the precision necessary for a given computational problem.

2.2 Using Arrays and Pointers

Optimization

Use array notation instead of pointer notation when working with arrays.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

C allows the use of either the array operator (`[]`) or pointers to access the elements of an array. However, the use of pointers in C makes work difficult for optimizers in C compilers. Without detailed and aggressive pointer analysis, the compiler has to assume that writes through a pointer can write to any location in memory, including storage allocated to other variables. (For example, `*p` and `*q` can refer to the same memory location, while `x[0]` and `x[2]` can't.) Using pointers causes aliasing, where the same block of memory is accessible in more than one way. Using array notation makes the task of the optimizer easier by reducing possible aliasing.

Example

Avoid code, such as the following, which uses pointer notation:

```
typedef struct {
   float x, y, z, w;
} VERTEX;

typedef struct {
   float m[4][4];
} MATRIX;

void XForm(float *res, const float *v, const float *m, int numverts) {

   float dp;
   int i;
   const VERTEX* vv = (VERTEX *)v;

   for (i = 0; i < numverts; i++) {
      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed x.

      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed y.

      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed z.

      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed w.

      ++vv;     // Next input vertex
      m -= 16;  // Reset to start of transform matrix.
   }
}
```

Instead, use the equivalent array notation:

```
typedef struct {
   float x, y, z, w;
} VERTEX;

typedef struct {
   float m[4][4];
} MATRIX;

void XForm(float *res, const float *v, const float *m, int numverts) {

   int i;
   const VERTEX* vv = (VERTEX *)v;
   const MATRIX* mm = (MATRIX *)m;
   VERTEX* rr = (VERTEX *)res;

   for (i = 0; i < numverts; i++) {
      rr->x = vv->x * mm->m[0][0] + vv->y * mm->m[0][1] +
              vv->z * mm->m[0][2] + vv->w * mm->m[0][3];
      rr->y = vv->x * mm->m[1][0] + vv->y * mm->m[1][1] +
              vv->z * mm->m[1][2] + vv->w * mm->m[1][3];
      rr->z = vv->x * mm->m[2][0] + vv->y * mm->m[2][1] +
              vv->z * mm->m[2][2] + vv->w * mm->m[2][3];
      rr->w = vv->x * mm->m[3][0] + vv->y * mm->m[3][1] +
              vv->z * mm->m[3][2] + vv->w * mm->m[3][3];
      ++rr;           // Increment the results pointer.
      ++vv;           // Increment the input vertex pointer.
   }
}
```

Additional Considerations

Source-code transformations interact with a compiler's code generator, making it difficult to control the generated machine code from the source level. It's even possible that source-code transformations aimed at improving performance may conflict with compiler optimizations. Depending on the compiler and the specific source code, it is possible for pointer-style code to compile into machine code that is faster than that generated from equivalent array-style code. Compare the performance of your code after implementing a source-code transformation with the performance of the original code to be sure that there is an improvement.

2.3 Unrolling Small Loops

Optimization

Completely unroll loops that have a small fixed loop count and a small loop body.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Many compilers don't aggressively unroll loops. Manually unrolling loops can benefit performance, especially if the loop body is small, which makes the loop overhead significant.

Example

Avoid a small loop like this:

```
// 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];
   }
}
```

Instead, replace it with its completely unrolled equivalent, as shown here:

```
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];
```

Related Information

For information on loop unrolling at the assembly-language level, see "Loop Unrolling" on page 145.

2.4 Expression Order in Compound Branch Conditions

Optimization

In the most active areas of a program, order the expressions in compound branch conditions to take advantage of short circuiting of compound conditional expressions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Branch conditions in C programs often consist of compound conditions consisting of multiple boolean expressions joined by the logical AND (&&) and logical OR (||) operators. C compilers guarantee short-circuit evaluation of these operators. In a compound logical OR expression, the first operand to evaluate to true terminates the evaluation, and subsequent operands are not evaluated at all. Similarly, in a logical AND expression, the first operand to evaluate to false terminates the evaluation. Because of this short-circuit evaluation, it is not always possible to swap the operands of logical OR and logical AND. This is especially true when the evaluation of one of the operands causes a side effect. However, in most cases the order of operands in such expressions is irrelevant.

When used to control conditional branches, expressions involving logical OR and logical AND are translated into a series of conditional branches. The ordering of the conditional branches is a function of the ordering of the expressions in the compound condition and can have a significant impact on performance. It is impossible to give an easy, closed-form formula on how to order the conditions. Overall performance is a function of a variety of the following factors:

- Probability of a branch misprediction for each of the branches generated
- Additional latency incurred due to a branch misprediction
- Cost of evaluating the conditions controlling each of the branches generated
- Amount of parallelism that can be extracted in evaluating the branch conditions
- Data stream consumed by an application (mostly due to the dependence of misprediction probabilities on the nature of the incoming data in data-dependent branches)

It is recommended to experiment with the ordering of expressions in compound branch conditions in the most active areas of a program (so-called “hot spots,” where most of the execution time is spent).

Such hot spots can be found through the use of profiling by feeding a typical data stream to the program while doing the experiments.

2.5 Long Logical Expressions in If Statements

Optimization

In `if` statements, avoid long logical expressions that can generate dense conditional branches that violate the guideline described in “Density of Branches” on page 126.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Listing 1. Preferred for Data that Falls Mostly Within the Range

```
if (a <= max && a >= min && b <= max && b >= min)
```

If most of the data falls within the range, the branches will not be taken, so the above code is preferred. Otherwise, the following code is preferred.

Listing 2. Preferred for Data that Does Not Fall Mostly Within the Range

```
if (a > max || a < min || b > max || b < min)
```

2.6 Arrange Boolean Operands for Quick Expression Evaluation

Optimization

In expressions that use the logical AND (&&) or logical OR (||) operator, arrange the operands for quick evaluation of the expression:

If the expression uses this operator	Then arrange the operands from left to right in decreasing probablity of being
&& (logical AND)	False
|| (logical OR)	True

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

C and C++ compilers guarantee short-circuit evaluation of the boolean operators && and ||. In an expression that uses &&, the first operand to evaluate to false terminates the evaluation; subsequent operands aren't evaluated. In an expression that uses ||, the first operand to evaluate to true terminates the evaluation.

When used to control program flow, expressions involving && and || are translated into a series of conditional branches. This optimization minimizes the total number of conditions evaluated and branches executed.

Example 1

In the following code, the operands of && aren't arranged for quick expression evaluation because the first operand isn't the condition most likely to be false (it is far less likely for an animal name to begin with a 'y' than for it to have fewer than four characters):

```
char animalname[30];
char *p;

p = animalname;

if ((strlen(p) > 4) && (*p == 'y')) { ... }
```

Because the odds that the animal name begins with a ‘y’ are comparatively low, it’s better to put that operand first:

```
if ((*p == 'y') && (strlen(p) > 4)) { ... }
```

Example 2

In the following code (assuming a uniform random distribution of `i`), the operands of `||` aren’t arranged for quick expression evaluation because the first operand isn’t the condition most likely to be true:

```
unsigned int i;

if ((i < 4) || (i & 1)) { ... }
```

Because it is more likely for the least-significant bit of `i` to be 1, it’s better to put that operand first:

```
if ((i & 1) || (i < 4)) { ... }
```

2.7 Dynamic Memory Allocation Consideration

Dynamic memory allocation—accomplished through the use of the `malloc` library function in C—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 that it is possible to cast the pointer to a `long`.

```
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. The pointer `p` is still needed in order to deallocate the storage.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

2.8 Unnecessary Store-to-Load Dependencies

A store-to-load dependency exists when data is stored to memory, only to be read back shortly thereafter. For details, see “Store-to-Load Forwarding Restrictions” on page 93. The AMD Athlon™ 64 and AMD Opteron™ processors contain 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 chain, as may 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. Therefore, it is recommended that the programmer remove the dependency manually, for example, by introducing a temporary variable that can be kept in a register, as in the following example. This can result in a significant performance increase.

Listing 3. 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]);
}
```

Listing 4. 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;
}
```

Application

This optimization applies to:

- 32-bit software
- 64-bit software

2.9 Matching Store and Load Size

Optimization

Align memory accesses and match addresses and sizes of stores and dependent loads.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The AMD Athlon 64 and AMD Opteron processors contain a load-store buffer to speed up the forwarding of store data to dependent loads. However, this store-to-load forwarding (STLF) inside the load-store buffer occurs, in general, only when the addresses and sizes of the store and the dependent load match, and when both memory accesses are aligned. For details, see “Store-to-Load Forwarding Restrictions” on page 93.

It is impossible to control load and store activity at the source level so as to avoid all cases that violate restrictions placed on store-to-load-forwarding. In some instances it is possible to spot such cases in the source code. Size mismatches can easily occur when different-size data items are joined in a union. Address mismatches could be the result of pointer manipulation.

The following examples show a situation involving a union of different-size data items. The examples show a user-defined unsigned 16.16 fixed-point type and two operations defined on this type. Function `fixed_add` adds two fixed-point numbers, and function `fixed_int` extracts the integer portion of a fixed-point number. Listing 5 shows an inappropriate implementation of `fixed_int`, which, when used on the result of `fixed_add`, causes misalignment, address mismatch, or size mismatch between memory operands, such that no store-to-load forwarding in the load-store buffer takes place. Listing 6 shows how to properly implement `fixed_int` in order to allow store-to-load forwarding in the load-store buffer.

Examples

Listing 5. Avoid

```
typedef union {
   unsigned int whole;
   struct {
      unsigned short frac; /* Lower 16 bits are fraction. */
      unsigned short intg; /* Upper 16 bits are integer.  */
   } parts;
} FIXED_U_16_16;
```

```
__inline FIXED_U_16_16 fixed_add(FIXED_U_16_16 x, FIXED_U_16_16 y) {
     FIXED_U_16_16 z;
     z.whole = x.whole + y.whole;
     return (z);
}

__inline unsigned int fixed_int(FIXED_U_16_16 x) {
     return((unsigned int)(x.parts.intg));
}
...
FIXED_U_16_16 y, z;
unsigned int q;
...
label1:
y = fixed_add (y, z);
q = fixed_int (y);

label2:
...
```

The object code generated for the source code between `label1` and `label2` typically follows one of these two variants:

```
; Variant 1
mov edx, DWORD PTR [z]
mov eax, DWORD PTR [y]     ; -+
add eax, edx               ;  |
mov DWORD PTR [y], eax     ;  |
mov EAX, DWORD PTR [y+2]   ; <+ Address mismatch--no forwarding in LSU
and EAX, 0FFFFh
mov DWORD PTR [q], eax

; Variant 2
mov   edx, DWORD PTR [z]
mov   eax, DWORD PTR [y]    ; -+
add   eax, edx              ;  |
mov   DWORD PTR [y], eax    ;  |
movzx eax, WORD PTR [y+2]   ; <+ Size and address mismatch--no forwarding in LSU
mov   DWORD PTR [q], eax
```

Listing 6. Preferred

```
typedef union {
   unsigned int whole;
   struct {
      unsigned short frac; /* Lower 16 bits are fraction. */
      unsigned short intg; /* Upper 16 bits are integer.  */
   } parts;
} FIXED_U_16_16;

__inline FIXED_U_16_16 fixed_add(FIXED_U_16_16 x, FIXED_U_16_16 y) {
```

```
   FIXED_U_16_16 z;
   z.whole = x.whole + y.whole;
   return(z);
}

__inline unsigned int fixed_int(FIXED_U_16_16 x) {
   return (x.whole >> 16);
}
...
FIXED_U_16_16 y, z;
unsigned int q;
...
label1:
y = fixed_add (y, z);
q = fixed_int (y);

label2:
...
```

The object code generated for the source code between `label1` and `label2` typically looks like this:

```
mov edx, DWORD PTR [z]
mov eax, DWORD PTR [y]
add eax, edx
mov DWORD PTR [y], eax   ; -+
mov eax, DWORD PTR [y]   ; <+ Aligned (size/address match)--forwarding in LSU
shr eax, 16
mov DWORD PTR [q], eax
```

2.10 SWITCH and Noncontiguous Case Expressions

Optimization

Use `if-else` statements in place of `switch` statements that have noncontiguous case expressions. (Case expressions are the individual expressions to which the single `switch` expression is compared.)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

If the case expressions are contiguous or nearly contiguous integer values, most compilers translate the `switch` statement as a jump table instead of a comparison chain. Jump tables generally improve performance because:

- They reduce the number of branches to a single procedure call.
- The size of the control-flow code is the same no matter how many cases there are.
- The amount of control-flow code that the processor must execute is the same for all values of the `switch` expression.

However, if the case expressions are noncontiguous values, most compilers translate the `switch` statement as a comparison chain. Comparison chains are undesirable because:

- They use dense sequences of conditional branches, which interfere with the processor's ability to successfully perform branch prediction.
- The size of the control-flow code increases with the number of cases.
- The amount of control-flow code that the processor must execute varies with the value of the `switch` expression.

Example 1

A `switch` statement like this one, whose case expressions are contiguous integer values, usually provides good performance:

```
switch (grade)
{
   case ‘A’:
      ...
      break;
   case ‘B’:
      ...
      break;
   case ‘C’:
      ...
      break;
   case ‘D’:
      ...
      break;
   case ‘F’:
      ...
      break;
}
```

Example 2

Because the case expressions in the following `switch` statement aren’t contiguous values, the compiler will likely translate the code into a comparison chain instead of a jump table:

```
switch (a)
{
   case 8:
      // Sequence for a==8
      break;
   case 16:
      // Sequence for a==16
      break;
   ...
   default:
      // Default sequence
      break;
}
```

To avoid a comparison chain and its undesirable effects on branch prediction, replace the `switch` statement with a series of `if-else` statements, as follows:

```
if (a==8) {
   // Sequence for a==8
}
else if (a==16) {
   // Sequence for a==16
}
...
else {
   // Default sequence
}
```

Related Information

For information on preventing branch-prediction interference at the assembly-language level, see “Density of Branches” on page 126.

2.11 Arranging Cases by Probability of Occurrence

Optimization

Arrange `switch` statement cases by probability of occurrence, from most probable to least probable.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Arranging `switch` statement cases by probability of occurrence improves performance when the `switch` statement is translated as a comparison chain; this arrangement has no negative impact when the statement is translated as a jump table.

Example

Avoid `switch` statements such as the following, in which the cases are not arranged by probability of occurrence:

```
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");
}
```

Instead, arrange the cases to test for frequently occurring values first:

```
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");
}
```

2.12 Use of Function Prototypes

Optimization

In general, use prototypes for all functions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Prototypes can convey additional information to the compiler that might enable more aggressive optimizations.

2.13 Use of const Type Qualifier

Optimization

For objects whose values will not be changed, use the `const` type qualifier.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using the `const` type qualifier makes code more robust and may enable the compiler to generate higher-performance code. For example, under the C standard, a compiler is not required to allocate storage for an object that is declared `const`, if its address is never used.

2.14 Generic Loop Hoisting

Optimization

To improve the performance of inner loops, reduce redundant constant calculations (that is, loop-invariant calculations). This idea can also be extended to invariant control structures.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale and Examples

The following example demonstrates the use of an invarient condition in an `if` statement in a `for` loop. The second listing shows the preferred optimization.

Listing 7. (Avoid)

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

Listing 8. (Preferred Optimzation)

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

The preferred optimization in Listing 8 tightens the inner loops 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 (which are usually advantageous) are saved.

To generalize the example in Listing 8 further for multiple-constant control code, more work may be needed to create the proper outer loop. Enumeration of the constant cases reduces this to a simple `switch` statement.

Listing 9.

```
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.
   }
}
```

Transform the loop in Listing 9 (by using the `switch` statement) 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;
}
```

Some introductory code is necessary to generate all the combinations for the `switch` constant and the total amount of code has doubled. However, the inner loops are now free of `if` statements. In ideal cases where the `DoWork`*n* functions are inlined, the successive functions have greater overlap, leading to greater parallelism than possible in the presence of intervening `if` statements.

The same idea can be applied to constant `switch` statements or to combinations of `switch` statements and `if` statements inside of `for` loops. The method used to combine the input constants becomes more complicated but benefits performance.

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 must be dealt with directly, and the remaining cases can fall back to the old code in the default clause of the `switch` statement. This situation is typical of run-time generated code. While the performance of run-time generated code can be improved by means similar to those presented here, it is much harder to maintain and developers must do their own code-generation optimizations without the help of an available compiler.

2.15 Local Static Functions

Optimization

Declare as `static` functions that are not used outside the file where they are defined.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Declaring a function as `static` forces internal linkage. Functions that aren't declared as `static` default to external linkage, which may inhibit certain optimizations—for example, aggressive inlining—with some compilers.

2.16 Explicit Parallelism in Code

Optimization

Where possible, break long dependency chains into several independent dependency chains that can then be executed in parallel, exploiting the execution units in each pipeline.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale and Examples

This is especially important to break long dependency chains into smaller executing units in floating-point code, whether it is mapped to x87, SSE, or SSE2 instructions, because of the longer latency of floating-point operations. Because most languages (including ANSI C) guarantee that floating-point expressions are not reordered, compilers cannot usually perform such optimizations unless they offer a switch to allow noncompliant reordering of floating-point expressions according to algebraic rules.

Reordered code that is algebraically identical to the original code does not necessarily produce 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. In the vast majority of cases, the final result differs only in the least-significant bits.

Listing 10. Avoid

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

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

Listing 11. 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 four-way unrolling is chosen to exploit the four-stage fully pipelined floating-point adder. Each stage of the floating-point adder is occupied on every clock cycle, ensuring maximum sustained utilization.

2.17 Extracting Common Subexpressions

Optimization

Manually extract common subexpressions where C compilers may be unable to extract them from floating-point expressions due to the guarantee against reordering of such expressions in the ANSI standard.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Specifically, the compiler cannot rearrange the computation according to algebraic equivalencies before extracting common subexpressions. Rearranging the expression may give different computational results due to the lack of associativity of floating-point operations, but the results usually differ in only the least-significant bits.

Examples

Listing 12. Avoid

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

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

Listing 13. Preferred

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

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

Listing 14. Avoid

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

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

Listing 15. Example 2 (Preferred)

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

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

2.18 Sorting and Padding C and C++ Structures

Optimization

Sort and pad C and C++ structures to achieve natural alignment.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In order to achieve better alignment for structures, many compilers have options that allow padding of structures to make their sizes multiples of words, doublewords, or quadwords. 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 implementations might not work properly in all situations.

By sorting and padding structures at the source-code level, if the first member of a structure is naturally aligned, then all other members are naturally aligned as well. This allows, for example, arrays of structures to be perfectly aligned.

Sorting and Padding C and C++ Structures

To sort and pad a C or C++ structure, follow these steps:

1. Sort the structure members according to their type sizes, declaring members with larger type sizes ahead of members with smaller type sizes.
2. Pad the structure so the size of the structure is a multiple of the largest member's type size.

Examples

Avoid structure declarations in which the members aren't declared in order of their type sizes and the size of the structure isn't a multiple of the size of the largest member's type:

```
struct {
   char a[5];   \\ Smallest type size (1 byte * 5)
   long k;      \\ 4 bytes in this example
   double x;    \\ Largest type size (8 bytes)
} baz;
```

Instead, declare the members according to their type sizes (largest to smallest) and add padding to ensure that the size of the structure is a multiple of the largest member's type size:

```
struct {
   double x;      \\ Largest type size (8 bytes)
   long k;        \\ 4 bytes in this example
   char a[5];     \\ Smallest type size (1 byte * 5)
   char pad[7];   \\ Make structure size a multiple of 8.
} baz;
```

2.19 Sorting Local Variables

Optimization

Sort local variables according to their type sizes, declaring those with larger type sizes ahead of those with smaller type sizes.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

It can be helpful to presort local variables, if your compiler allocates local variables in the same order in which they are declared in the source code. If the first variable is allocated for natural alignment, all other variables are allocated contiguously in the order they are declared and are naturally aligned without padding.

Some compilers do not allocate variables in the order they are declared. In these cases, the compiler should automatically allocate variables that are naturally aligned with the minimum amount of padding. In addition, some compilers do not guarantee that the stack is aligned suitably for the largest 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.

Example

Avoid local variable declarations, when the variables aren't declared in order of their type sizes:

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

Instead, sort the declarations according to their type sizes (largest to smallest):

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

Related Information

For information on sorting local variables at the assembly-language level, see “Sorting Local Variables” on page 111.

2.20 Replacing Integer Division with Multiplication

Optimization

Replace integer division with multiplication when there are multiple divisions in an expression. (This is possible only if no overflow will occur during the computation of the product. The possibility of an overflow can be determined by considering the possible ranges of the divisors.)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Integer division is the slowest of all integer arithmetic operations.

Examples

Avoid code that uses two integer divisions:

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

m = i / j / k;
```

Instead, replace one of the integer divisions with the appropriate multiplication:

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

2.21 Frequently Dereferenced Pointer Arguments

Optimization

Avoid dereferenced pointer arguments inside a function.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Because the compiler has no knowledge of whether aliasing exists between the pointers, such dereferencing cannot be "optimized away" by the compiler. Since data may not be maintained in registers, memory traffic can significantly increase.

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. If your compiler doesn't have this type of optimization, then 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.

Examples

Listing 16. Avoid

```
// Assumes pointers are different and q != r.
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;
}
```

Listing 17. Preferred

```
// Assumes pointers are different and q != r.
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;
}
```

2.22 Using Signed Integers for 32-Bit Array Indices

Optimization

When using a 32-bit variable as an array index, declare the variable as an unsigned integer instead of a signed integer.

Application

This optimization applies to 64-bit software.

Rationale

When performing 64-bit address arithmetic, the compiler may be able to generate more efficient code if signed 32-bit integers are used as array indices, instead of unsigned 32-bit integers. (There is no performance penalty for using 64-bit variables—either signed or unsigned—as array indices.)

2.23 32-Bit Integral Data Types

Optimization

Use 32-bit integers instead of integers with smaller sizes (16-bit or 8-bit).

Application

This optimization applies to 32-bit software.

Rational

Be aware of the amount of storage associated with each integral data type.

2.24 Sign of Integer Operands

Optimization

Where there is a choice of using either a signed or an unsigned type, take into consideration that some operations are faster with unsigned types while others are faster for signed types.

Application

This optimization applies to:

- 32-bit software

Rationale

In many cases, the type of data to be stored in an integer variable determines whether a signed or an unsigned integer type is appropriate. For example, to record the weight of a person in pounds, no negative numbers are required, so an unsigned type is appropriate. However, recording temperatures in degrees Celsius may require both positive and negative numbers, so a signed type is needed.

Integer-to-floating-point conversion using integers larger than 16 bits is faster with signed types, as the AMD64 architecture provides instructions for converting signed integers to floating-point but has no instructions for converting unsigned integers. In a typical case, a 32-bit integer is converted by a compiler to assembly as follows:

Examples

Listing 18. (Avoid)

```
double x;          ====>   mov [temp+4], 0
unsigned int i;            mov eax, i
                           mov [temp], eax
x = i;                     fild QWORD PTR [temp]
                           fstp QWORD PTR [x]
```

The preceding code is slow not only because of the number of instructions, but also because a size mismatch prevents store-to-load forwarding to the FILD instruction. Instead, use the following code:

Listing 19. (Preferred)

```
double x;   ====>   fild DWORD PTR [i]
int i;              fstp QWORD PTR [x]

x = i;
```

Computing quotients and remainders in integer division by constants is faster when performed on unsigned types. The following typical case is the compiler output for a 32-bit integer divided by 4:

Listing 20. Example 2 (Avoid)

```
int i;       ====>   mov eax, i
                     cdq
i = i / 4;           and edx, 3
                     add eax, edx
                     sar eax, 2
                     mov i, eax
```

Listing 21. Example 2 (Preferred)

```
unsigned int i;   ====>   shr  i, 2

i = i / 4;
```

In summary, use unsigned types for:

- Division and remainders
- Loop counters
- Array indexing

Use signed types for:

- Integer-to-floating-point conversion

2.25 Accelerating Floating-Point Division and Square Root

Optimization

In applications that involve the heavy use of single precision division and square root operations, it is recommended that you port the code to SSE or 3DNow!™ inline assembly or use a compiler that can generate SSE or 3DNow! technology code. If neither of these methods are possible, the x87 FPU control word register precision control specification bits (PC) can be set to single precision to improve performance. (The processor defaults to double-extended precision. See *AMD64 Architecture Programmer's Manual Volume 1: Application Programming* (order# 24592) for details on the FPU control register.)

Application

This optimization applies to 32-bit software.

Rationale

Division and square root have a much longer latency than other floating-point operations, even though the AMD Athlon 64 and AMD Opteron processors provide significant acceleration of these two operations. In some application programs, these operations occur so often as to seriously impact performance. If code has hot spots that use single precision arithmetic only (that is, all computation involves data of type `float`) and for some reason cannot be ported to 3DNow! code, the following technique may be used to improve performance.

The x87 FPU has a precision-control field as part of the FPU control word. The precision-control setting determines rounding precision of instruction results and affects the basic arithmetic operations, including division and the extraction of square root. Division and square root on the AMD Athlon 64 and AMD Opteron processors are only computed to the number of bits necessary for the currently selected precision. Setting precision control to single precision (versus the Win32 default of double precision) lowers the latency of those operations.

The Microsoft® Visual C environment provides functions to manipulate the FPU control word and thus the precision control. Note that these functions are not very fast, so insert changes of precision control where it creates little overhead, such as outside a computation-intensive loop. Otherwise, the overhead created by the function calls outweighs the benefit from reducing the latencies of divide and square-root operations. For more information on this topic, see *AMD64 Architecture Programmer's Manual Volume 1: Application Programming* (order# 24592).

The following example shows how to set the precision control to single precision and later restore the original settings in the Microsoft Visual C environment.

Examples

Listing 22.

```
/* Prototype for _controlfp function */
#include <float.h>
unsigned int orig_cw;

/* Get current FPU control word and save it. */
orig_cw = _controlfp(0, 0);

/* Set precision control in FPU control word to single precision.
This reduces the latency of divide and square-root operations. */
_controlfp(_PC_24, MCW_PC);

/* Restore original FPU control word. */
_controlfp(orig_cw, 0xfffff);
```

2.26 Fast Floating-Point-to-Integer Conversion

Optimization

Use 3DNow! PF2ID instruction to perform truncating conversion to accomplish rapid floating-point-to-integer conversion, if the floating-point operand is a type `float`.

Application

This optimization applies to 32-bit software.

Rationale

Floating-point-to-integer conversion in C programs is typically a very slow operation. The semantics of C and C++ demand that the conversion use truncation. If the floating-point operand is of type `float`, and the compiler supports 3DNow! code generation, then the 3DNow! PF2ID instruction, which performs truncating conversion, can be utilized by the compiler to accomplish rapid floating-point-to-integer conversion.

Note: *The PF2ID instruction does not provide conversion compliant with the IEEE-754 standard. Some operands of type* `float` *(IEEE-754 single precision) such as NaNs, infinities, and denormals, are either unsupported or not handled in compliance with the IEEE-754 standard by 3DNow! technology.*

For double precision operands, the usual way to accomplish truncating conversion involves the following algorithm:

1. Save the current x87 rounding mode (this is usually round to nearest or even).
2. Set the x87 rounding mode to truncation.
3. Load the floating-point source operand and store the integer result.
4. Restore the original x87 rounding mode.

This algorithm is typically implemented through the C run-time library function `ftol`. While the AMD Athlon 64 and AMD Opteron processors have special hardware optimizations to speed up the changing of x87 rounding modes and therefore `ftol`, calls to `ftol` may still tend to be slow.

For situations where very fast floating-point-to-integer conversion is required, the conversion code in Listing 24 on page 53 may be helpful. This code uses the current rounding mode instead of truncation when performing the conversion. Therefore, the result may differ by 1 from the `ftol` result. The replacement code adds the "magic number" $2^{52}+2^{51}$ to the source operand, then stores the double precision result to memory and retrieves the lower doubleword of the stored result. Adding the magic number shifts the original argument to the right inside the double precision mantissa, placing the binary point of the sum immediately to the right of the least-significant mantissa bit. Extracting the lower doubleword of the sum then delivers the integral portion of the original argument.

The following conversion code causes a 64-bit store to feed into a 32-bit load. The load is from the lower 32 bits of the 64-bit store, the one case of size mismatch between a store and a dependent load that is specifically supported by the store-to-load-forwarding hardware of the AMD Athlon 64 and AMD Opteron processors.

Examples

Listing 23. Slow

```
double x;
int i;

i = x;
```

Listing 24. Fast

```
#define DOUBLE2INT(i, d) \
   {double t = ((d) + 6755399441055744.0); i = *((int *)(&t));}

double x;
int i;

DOUBLE2INT(i, x);
```

2.27 Speeding Up Branches Based on Comparisons Between Floats

Optimization

Store operands of type float into a memory location and use integer comparison with the memory location to perform fast branches in cases where compilers do not support fast floating-point comparison instructions or 3DNow! code generation.

Application

This optimization applies to 32-bit software.

Rationale

Branches based on floating-point comparisons are often slow. The AMD Athlon 64 and AMD Opteron processors support the FCOMI, FUCOMI, FCOMIP, and FUCOMIP instructions that allow implementation of fast branches based on comparisons between operands of type `double` or type `float`. However, many compilers do not support generating these instructions. Likewise, floating-point comparisons between operands of type `float` can be accomplished quickly by using the 3DNow! PFCMP instruction if the compiler supports 3DNow! code generation.

Many compilers only implement branches based on floating-point comparisons by using FCOM or FCOMP to compare the floating-point operands, followed by `FSTSW AX` in order to transfer the x87 condition-code flags into EAX. The subsequent branch is then based on the contents of the EAX register. Although the AMD Athlon 64 and AMD Opteron processors have acceleration hardware to speed up the FSTSW instruction, this process is still fairly slow.

Branches Dependent on Integer Comparisons Are Fast

One alternative for branches dependent upon the outcome of the comparison of operands of type `float` is to store the operand(s) into a memory location and then perform an integer comparison with that memory location. Branches dependent on integer comparisons are very fast. It should be noted that the replacement code uses a load dependent on an immediately prior store. If the store is not doubleword-aligned, no store-to-load-forwarding takes place, and the branch is still slow. Also, if there is a lot of activity in the load-store queue, forwarding of the store data may be somewhat delayed, thus negating some of the advantages of using the replacement code. It is recommended that you experiment with the replacement code to test whether it actually provides a performance increase in the code at hand.

The replacement code works well for comparisons against zero, including correct behavior when encountering a negative zero as allowed by the IEEE-754 standard. It also works well for comparing

to positive constants. In that case, the user must first determine the integer representation of that floating-point constant. This can be accomplished with the following C code snippet:

```
float x;
scanf("%g", &x);
printf("%08X\n", (*((int *)(&x))));
```

The replacement code is IEEE-754 compliant for all classes of floating-point operands except NaNs. However, NaNs do not occur in properly working software.

Examples

Intial definitions:

```
#define FLOAT2INTCAST(f)  (*((int *)(&f)))
#define FLOAT2UINTCAST(f) (*((unsigned int *)(&f)))
```

Table 3: Comparisons against Zero

Use this ...	Instead of this.
`if (FLOAT2UINTCAST(f) > 0x80000000U)`	`if (f < 0.0f)`
`if (FLOAT2INCAST(f) <= 0)`	`if (f <= 0.0f)`
`if (FLOAT2INTCAST(f) > 0)`	`if (f > 0.0f)`
`if (FLOAT2UINTCAST(f) <= 0x80000000U)`	`if (f >= 0.0f)`

Table 4: Comparisons against Positive Constant

Use this ...	Instead of this.
`if (FLOAT2INTCAST(f) < 0x40400000)`	`if (f < 3.0f)`
`if (FLOAT2INTCAST(f) <= 0x40400000)`	`if (f <= 3.0f)`
`if (FLOAT2INTCAST(f) > 0x40400000)`	`if (f > 3.0f)`
`if (FLOAT2INTCAST(f) >= 0x40400000)`	`if (f >= 3.0f)`

Table 5: Comparisons among Two Floats

Use this ...	Instead of this.
`float t = f1 - f2;` `if (FLOAT2UINTCAST(t) > 0x80000000U)`	`if (f1 < f2)`
`float t = f1 - f2;` `if (FLOAT2INTCAST(t) <= 0)`	`if (f1 <= f2)`
`float t = f1 - f2;` `if (FLOAT2INTCAST(t) > 0)`	`if (f1 > f2)`
`float t = f1 - f2;` `f (FLOAT2UINTCAST(f) <= 0x80000000U)`	`if (f1 >= f2)`

Chapter 3 General 64-Bit Optimizations

In long mode, the AMD64 architecture provides both a compatibility mode, which allows a 64-bit operating system to run existing 16-bit and 32-bit applications, and a 64-bit mode, which provides 64-bit addressing and expanded register resources to support higher performance for recompiled 64-bit programs. This chapter presents general optimizations that improve the performance of software designed to run in 64-bit mode. Therefore, all optimizations in this chapter apply only to 64-bit software.

This chapter covers the following topics:

3.1 64-Bit Registers and Integer Arithmetic

Optimization

Use 64-bit registers for 64-bit integer arithmetic.

Rationale

Using 64-bit registers instead of their 32-bit equivalents can dramatically reduce the amount of code necessary to perform 64-bit integer arithmetic.

Example 1

This code performs 64-bit addition using 32-bit registers:

```
; Add ECX:EBX to EDX:EAX, and place sum in EDX:EAX.
00000000  03 C3  add  eax, ebx
00000002  13 D1  adc  edx, ecx
```

Using 64-bit registers, the previous code can be replaced by one simple instruction (assuming that RAX and RBX contain the 64-bit integer values to add):

```
00000000  48 03 C3  add  rax, rbx
```

Although the preceding instruction requires one additional byte for the REX prefix, it's still one byte shorter than the original code. More importantly, this instruction still has a latency of only one cycle, uses two fewer registers, and occupies only one decode slot.

Example 2

To perform the low-order half of the product of two 64-bit integers using 32-bit registers, a procedure such as the following is necessary:

```
; In:       [ESP+8]:[ESP+4] = multiplicand
;           [ESP+16]:[ESP+12] = multiplier
; Out:      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 + p2_lo + p3_lo = result in EDX:EAX
   ret                       ; Done, return to caller.

llmul ENDP
```

Using 64-bit registers, the entire product can be produced with only one instruction:

```
; Multiply RAX by RBX. The 128-bit product is stored in RDX:RAX.
00000000  48 F7 EB  imul  rbx
```

Related Information

For more examples of 64-bit arithmetic using only 32-bit registers, see “Efficient 64-Bit Integer Arithmetic in 32-Bit Mode” on page 170.

3.2 64-Bit Arithmetic and Large-Integer Multiplication

Optimization

Use 64-bit arithmetic for integer multiplication that produces 128-bit or larger products.

Background

Large-number multiplications (those involving 128-bit or larger products) are utilized in cryptography algorithms, which figure importantly in e-commerce applications and secure transactions on the Internet. Processors cannot perform large-number multiplication natively; they must break the operation into chunks that are permitted by their architecture (32-bit or 64-bit additions and multiplications).

Rationale

Using 64-bit rather than 32-bit integer operations dramatically reduces the number of additions and multiplications required to compute large products. For example, computing a 1024-bit product using 64-bit arithmetic requires fewer than one quarter the number of instructions required when using 32-bit operations:

Comparing...	32-bit arithmetic	64-bit arithmetic
Number of multiplications	256	64
Number of additions with carry	509	125
Number of additions	255	63

In addition, the processor performs 64-bit additions just as fast as it performs 32-bit additions, and the latency of 64-bit multiplications is only slightly higher than for 32-bit multiplications. (The processor is capable of performing a 64-bit addition each clock cycle and a 64-bit multiplication every other clock cycle.)

Example

Consider the multiplication of two unsigned 64-bit numbers *a* and *b*, represented in terms of 32-bit numbers *a1*:*a0* and *b1*:*b0*.

$a = a1 * 2^{32} + a0$

$b = b1 * 2^{32} + b0$

The product of *a* and *b*, *c*, can be expressed in terms of products of the 32-bit components, as follows:

$c = (a1 * b1) * 2^{64} + (a1 * b0 + a0 * b1) * 2^{32} + (a0 * b0)$

Each of the products of the components of *a* and *b* (for example, *a1 * b1*) is composed of 64 bits—an upper 32 bits and a lower 32 bits. It's convenient to represent these individual products as *d*, *e*, *f*, and *g*, as follows:

$a0 * b0 = d1{:}d0 = d1 * 2^{32} + d0$

$a1 * b0 = e1{:}e0 = e1 * 2^{32} + e0$

$a0 * b1 = f1{:}f0 = f1 * 2^{32} + f0$

$a1 * b1 = g1{:}g0 = g1 * 2^{32} + g0$

Substitution yields the following equation:

$c = (g1 * 2^{32} + g0) * 2^{64} + (e1 * 2^{32} + e0 + f1 * 2^{32} + f0) * 2^{32} + (d1 * 2^{32} + d0)$

Simplifying yields this equation:

$c = g1 * 2^{96} + (e1 + f1 + g0) * 2^{64} + (d1 + e0 + f0) * 2^{32} + d0$

It's convenient to represent the terms that are multiplied by each power of 2 as *c3*, *c2*, *c1*, and *c0*, as follows:

$g1 = c3$

$e1 + f1 + g0 = c2$

$d1 + e0 + f0 = c1$

$d0 = c0$

Substituting again yields:

$c = c3 * 2^{96} + c2 * 2^{64} + c1 * 2^{32} + c0$

The following procedure performs 64-bit unsigned integer multiplication, as previously illustrated using 32-bit integer operations:

```
; 32bitalu_64x64(int *a, int *b, int *c);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c 32bitalu_64x64.asm
;
.586
.K3D
.XMM
_DATA   SEGMENT
tempESP dd 0
_DATA   ENDS
_TEXT   SEGMENT
ASSUME DS:_DATA
PUBLIC _32bitalu_64x64
_32bitalu_64x64 PROC NEAR
;=============================================================================
; Save the register state. Registers EAX, ECX, and EDX are considered volatile
;  and assumed to be changed, while the registers below must be preserved.
push ebp
mov  ebp, esp
;=============================================================================
; Parameters passed into routine:
;  [ebp+8]   = ->a
;  [ebp+12]  = ->b
;  [ebp+16]  = ->c
;=============================================================================
push ebx
push esi
push edi
;=============================================================================
mov  esi,[ebp+8]      ; ESI = ->a
mov  edi,[ebp+12]     ; EDI = ->b
mov  ecx,[ebp+16]     ; ECX = ->c
push ebp
mov  [tempESP], esp
;=============================================================================
; Multiply 64-bit numbers a and b, each of which is composed of two 32-bit
;  components:
;  a = a1 * 2^32 + a0
;  b = b1 * 2^32 + b0
mov eax,[esi]      ; EAX = a0
mov edx,[edi]      ; EDX = b0
mul edx            ; EDX:EAX = a0*b0 = d1:d0
mov ebx,edx        ; EDX = d1
mov [ecx],eax      ; c0 = EAX

xor esp,esp        ; ESP = 0
xor ebp,ebp        ; EBP = 0
```

```
mov eax,[esi+4]    ; EAX = a1
mov edx,[edi]      ; EDX = b0
mul edx            ; EDX:EAX = a1*b0 = e1:e0
add ebx,eax        ; EBX = d1 + e0
adc ebp,edx        ; EBP = e1 + possible carry from d1+e0
adc esp,0          ; Collect possible carry into c3.

mov eax,[esi]      ; EAX = a0
mov edx,[edx+4]    ; EDX = b1
mul edx            ; EDX:EAX = a0*b1 = f1:f0
add ebx,eax        ; EBX = d1 + e0 + f0
adc ebp,edx        ; EBP = e1 + f1 + carry
adc esp,0          ; Collect possible carry into c3.
mov [ecx+4],ebx    ; c1 = d1 + e0 + f0

mov eax,[esi+4]    ; EAX = a1
mov edx,[edi+4]    ; EDX = b1
mul edx            ; EDX:EAX = a1*b1 = g1:g0
add ebp,eax        ; EBP = e1 + f1 + g0 + carry
adc esp,edx        ; ESP = g1 + carry
mov [ecx+8],ebp    ; c2 = e1 + f1 + g0 + carry
mov [ecx+12],esp   ; c3 = g1 + carry
;============================================================================
; Restore the register state.
mov esp, [tempESP]
pop ebp
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
;============================================================================
ret
_32bitalu_64x64 ENDP
_TEXT   ENDS
END
```

To improve performance and substantially reduce code size, the following procedure performs the same 64-bit integer multiplication using 64-bit instead of 32-bit operations:

```
; 64bitalu_64x64(int *a, int *b, int *c);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml64.exe -c 64bitalu_64x64.asm
;
_TEXT   SEGMENT
64bitalu_64x64 PROC NEAR
;=============================================================================
; Parameters passed into routine:
;  rcx = ->a
;  rdx = ->b
;  r8  = ->c
;=============================================================================
mov rax, [rcx]    ; RAX = [a0]
mul [rdx]         ; Multiply [a0] by [b0] such that
                  ;  RDX:RAX = [c1]:[c0].
mov [r8], rax     ; Store 128-bit product of a and b.
mov [r8+8], rdx
;=============================================================================
ret
64bitalu_64x64 ENDP
END
```

3.3 128-Bit Media Instructions and Floating-Point Operations

Optimization

Use 128-bit media (SSE and SSE2) instructions instead of x87 or 64-bit media (MMX™ and 3DNow!™ technology) instructions for floating-point operations.

Rationale

In 64-bit mode, the processor provides eight additional XMM registers (XMM8–XMM15) for a total of 16. These extra registers can substantially reduce register pressure in floating-point code written using 128-bit media instructions.

Although the processor fully supports the x87 and 64-bit media instructions, there are only eight registers available to these instructions (ST(0)–ST(7) or MMX0–MMX7, respectively).

3.4 32-Bit Legacy GPRs and Small Unsigned Integers

Optimization

Use the 32-bit legacy general-purpose registers (EAX through ESI) instead of their 64-bit extensions to store unsigned integer values whose range never requires more than 32 bits, even if subsequent statements use the 32-bit value in a 64-bit operation. (For example, use ECX instead of RCX until you need to perform a 64-bit operation; then use RCX.)

Rationale

In 64-bit mode, the machine-language representation of many instructions that operate on 64-bit register operands requires a REX prefix byte, which increases the size of the code. However, instructions that operate on a 32-bit legacy register operand don't require the prefix and have the desirable side-effect of clearing the upper 32 bits of the extended register to zero. For example, using the AND instruction on ECX clears the upper half of RCX.

Caution

Because the assembler also uses a REX prefix byte to encode the 32-bit sizes of the eight new 64-bit general-purpose registers (R8D–R15D), you should only use one of the original eight general-purpose registers (EAX through ESI) to implement this technique.

Example

The following example illustrates the unnecessary use of 64-bit registers to calculate the number of bytes remaining to be copied by an aligned block-copy routine after copying the first few bytes having addresses not meeting the routine's 8-byte-alignment requirements. The first two statements, after the program comments, use the 64-bit R10 register—presumably, because this value is later used to adjust a 64-bit value in R8—even though the range of values stored in R10 take no more than four bits to represent. Using R10 instead of a smaller register requires a REX prefix byte (in this case, 49), which increases the size of the machine-language code.

```
; Input:
;  R10 = source address (src)
;  R8 = number of bytes to copy (count)
49 F7 DA      neg r10       ; Subtract the source address from 2^64.
49 83 E2 07   and r10, 7    ; Determine how many bytes were copied separately.
4D 2B C2      sub r8, r10   ; Subtract the number of bytes already copied from
                            ;  the number of bytes to copy.
```

To improve code density, the following rewritten code uses ECX until it's absolutely necessary to use RCX, eliminating two REX prefix bytes:

```
F7 D9       neg ecx         ; Subtract the source address from 2^32 (the processor
                            ; clears the high 32 bits of RCX).
83 E1 07    and ecx, 7      ; Determine how many bytes were copied separately.
4C 2B C1    sub r8, rcx     ; Subtract the number of bytes already copied from
                            ; the number of bytes to copy.
```

Chapter 4 Instruction-Decoding Optimizations

The optimizations in this chapter are designed to help maximize the number of instructions that the processor can decode at one time.

The instruction fetcher of both the AMD Athlon™ 64 and AMD Opteron™ processors reads 16-byte packets from the L1 instruction cache. These packets are 16-byte aligned. The instruction bytes are then merged into a 32-byte pick window. On each cycle, the in-order front-end engine selects for decode up to three AMD64 instructions from the pick window.

This chapter covers the following topics:

4.1 DirectPath Instructions

Optimization

❖Use DirectPath instructions rather than VectorPath instructions. (To determine the type of an instruction—either DirectPath or VectorPath—see Appendix C, “Instruction Latencies.”)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

DirectPath instructions minimize the number of operations per AMD64 instruction, thus providing for optimally efficient decode and execution. Up to three DirectPath single instructions, or one and a half DirectPath double instructions, can be decoded per cycle. VectorPath instructions block the decoding of DirectPath instructions.

The AMD Athlon 64 and AMD Opteron processors implement the majority of instructions used by a compiler as DirectPath Single and DirectPath Double instructions. However, assembly writers must still take into consideration the use of DirectPath versus VectorPath instructions.

4.2 Load-Execute Instructions

A *load-execute instruction* is an instruction that loads a value from memory into a register and then performs an operation on that value. Many general purpose instructions, such as ADD, SUB, AND, etc., have load-execute forms:

```
add rax, QWORD PTR [foo]
```

This instruction loads the value `foo` from memory and then adds it to the value in the RAX register.

The work performed by a load-execute instruction can also be accomplished by using two discrete instructions—a load instruction followed by an execute instruction. The following example employs discrete load and execute stages:

```
mov rbx, QWORD PTR [foo]
add rax, rbx
```

The first statement loads the value `foo` from memory into the RBX register. The second statement adds the value in RBX to the value in RAX.

The following optimizations govern the use of load-execute instructions:

- Load-Execute Integer Instructions on page 71.
- Load-Execute Floating-Point Instructions with Floating-Point Operands on page 72.
- Load-Execute Floating-Point Instructions with Integer Operands on page 72.

4.2.1 Load-Execute Integer Instructions

Optimization

❖When performing integer computations, use load-execute instructions instead of discrete load and execute instructions. Use discrete load and execute instructions only to avoid scheduler stalls for longer executing instructions and to explicitly schedule load and execute operations.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

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 reduces decoding bandwidth and increases register pressure, which results in lower performance.

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

Optimization

❖When performing floating-point computations using floating-point (not integer) source operands, use load-execute instructions instead of discrete load and execute instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using load-execute floating-point instructions that take floating-point operands improves performance for the following reasons:

- Denser code allows more work to be held in the instruction cache.
- Denser code generates fewer internal macro-ops, allowing the floating-point scheduler to hold more work, which increases the chances of extracting parallelism from the code.

Example

Avoid code like this, which uses discrete load and execute instructions:

```
movss xmm0, [float_var1]
movss xmm12, [float_var2]
mulss xmm0, xmm12
```

Instead, use code like this, which uses a load-execute floating-point instruction:

```
movss xmm0, [float_var1]
mulss xmm0, [float_var2]
```

4.2.3 Load-Execute Floating-Point Instructions with Integer Operands

Optimization

❖Avoid x87 load-execute floating-point instructions that take integer operands (FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FIMUL, FISUB, and FISUBR). When performing floating-point computations using integer source operands, use discrete load (FILD) and execute instructions instead.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The load-execute floating-point instructions that take integer operands are VectorPath instructions and generate two micro-ops in a cycle, while discrete load and execute intructions enable a third DirectPath instruction to be decoded in the same cycle. In some situations, these optimizations can also reduce execution time if FILD can be scheduled several instructions ahead of the arithmetic instruction in order to cover the FILD latency.

Example

Avoid code such as the following, which uses load-execute floating-point instructions that take integer operands:

```
fld   QWORD PTR [foo]   ; Push foo onto FP stack [ST(0) = foo].
fimul DWORD PTR [bar]   ; Multiply bar by ST(0) [ST(0) = bar * foo].
fiadd DWORD PTR [baz]   ; Add baz to ST(0) [ST(0) = baz + (bar * foo)].
```

Instead, use code such as the following, which uses discrete load and execute instructions:

```
fild  DWORD PTR [bar]   ; Push bar onto FP stack.
fild  DWORD PTR [baz]   ; Push baz onto FP stack.
fld   QWORD PTR [foo]   ; Push foo onto FP stack.
fmulp st(2), st         ; Multiply and pop [ST(1) = foo * bar, ST(0) = baz].
faddp st(1), st         ; Add and pop [ST(0) = baz + (foo * bar)].
```

4.3 Branch Targets in Program Hot Spots

Optimization

In program “hot spots” (as determined by either profiling or loop-nesting analysis), branch targets should be placed at or near the beginning of code windows that are 16-byte aligned. The smaller the basic block, the more beneficial this optimization will be.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Aligning branch targets maximizes the number of instructions in the pick window and preserves instruction-cache space in branch-intensive code outside such hot spots.

4.4 32/64-Bit vs. 16-Bit Forms of the LEA Instruction

Optimization

Use the 32-bit or 64-bit forms of the Load Effective Address (LEA) instruction rather than the 16-bit form.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The 32-bit and 64-bit LEA instructions are implemented as DirectPath operations with an execution latency of only two cycles. The 16-bit LEA instruction, however, is a VectorPath instruction, which lowers the decode bandwidth and has a longer execution latency.

4.5 Short Instruction Encodings

Optimization

Use instruction forms with shorter encodings rather than those with longer encodings. For example, use 8-bit displacements instead of 32-bit displacements, and use the single-byte form of simple integer instructions instead of the 2-byte opcode-ModRM form.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using shorter instructions increases the number of instructions that can fit into the L1 instruction cache and increases the average decode rate.

Example

Avoid the use of instructions with longer encodings, such as those shown here:

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

Instead, choose instructions with shorter encodings, like these:

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

4.6 Partial-Register Reads and Writes

Optimization

Avoid partial register reads and writes.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In order to handle partial register writes, the processor's execution core implements a data merging scheme.

In the execution unit, an instruction that writes part of a register merges the modified portion with the current state of the other part 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.

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 1

Avoid code such as the following, which writes to only part of a register:

```
mov al, 10   ; Instruction 1
mov ah, 12   ; Instruction 2 has a false dependency on instruction 1.
             ; Instruction 2 merges new AH with current EAX register
             ;  value forwarded by instruction 1.
```

Example 2

Avoid code such as the following, which both reads and writes only parts of registers:

```
mov bx, 12h   ; Instruction 1
mov bl, dl    ; Instruction 2 has a false dependency on the completion
              ;  of instruction 1.
mov bh, cl    ; Instruction 3 has a false dependency on the completion
              ;  of instruction 2.
mov al, bl    ; Instruction 4 depends on the completion of instruction 2.
```

Example 3

Avoid:

```
mov    al, bl
```

Preferred:

```
movzx  eax, bl
```

Example 4

Avoid:

```
mov    al, [ebx]
```

Preferred:

```
movzx  eax, byte ptr [ebx]
```

Example 5

Avoid:

```
mov    al, 01h
```

Preferred:

```
mov    eax, 00000001h
```

Example 6

Avoid:

```
movss    xmm1, xmm2
```

Preferred:

```
movaps   xmm1, xmm2
```

4.7 Using LEAVE for Function Epilogues

Optimization

The recommended optimization for function epilogues depends on whether the function allocates local variables.

If the function	Then
Allocates local variables	Replace the traditional function epilogue with the LEAVE instruction.
Doesn't allocate local variables	Don't use function prologues or epilogues. Access function arguments and local variables through rSP.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Functions That Allocate Local Variables

The LEAVE instruction is a single-byte instruction and saves 2 bytes of code space over the traditional epilogue. Replacing the traditional sequence with LEAVE also preserves decode bandwidth.

Functions That Don't Allocate Local Variables

Accessing function arguments and local variables directly through ESP frees EBP for use as a general-purpose register.

Background

The function arguments and local variables inside a function are referenced through a so-called frame pointer. In AMD64 code, the base pointer register (rBP) is customarily used as a frame pointer. You set up the frame pointer at the beginning of the function using a function prologue:

```
push ebp                    ; Save old frame pointer.
mov  ebp, esp               ; Initialize new frame pointer.
sub  esp, n                 ; Allocate space for local variables (only if the
                            ; function allocates local variables).
```

Function arguments on the stack can now be accessed at positive offsets relative to rBP, and local variables are accessible at negative offsets relative to rBP.

Example

The traditional function epilogue looks like this:

```
mov esp, ebp   ; Deallocate local variables (only if space was allocated).
pop ebp        ; Restore old frame pointer.
```

Replace the traditional function epilogue with a single LEAVE instruction:

```
leave
```

4.8 Alternatives to SHLD Instruction

Optimization

Where register pressure is low, replace the SHLD instruction with alternative code using ADD and ADC, or SHR and LEA.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using alternative code in place of SHLD achieves lower overall latency and requires fewer execution resources. The 32-bit and 64-bit forms of ADD, ADC, SHR, and LEA are DirectPath instructions, while SHLD is a VectorPath instruction. Use of the replacement code optimizes decode bandwidth because it potentially enables the simultaneous decoding of a third DirectPath instruction. However, the replacement code may increase register pressure because it destroys the contents of one register (*reg2* in the following examples) whereas the register is preserved by SHLD.

Example 1

Replace this instruction:

```
shld reg1, reg2, 1
```

with this code sequence:

```
add reg2, reg2
adc reg1, reg1
```

Example 2

Replace this instruction:

```
shld reg1, reg2, 2
```

with this code sequence:

```
shr reg2, 30
lea reg1, [reg1*4+reg2]
```

Example 3

Replace this instruction:

```
shld reg1, reg2, 3
```

with this code sequence:

```
shr reg2, 29
lea reg1, [reg1*8+reg2]
```

4.9 8-Bit Sign-Extended Immediate Values

Optimization

Use 8-bit sign-extended immediate values instead of larger-size values.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using 8-bit sign-extended immediate values improves code density with no negative affects on the processor.

Example

Consider this instruction:

```
add bx, -5
```

Avoid encoding it as:

```
81 C3 FF FB
```

Instead, encode it as:

```
83 C3 FB
```

4.10 8-Bit Sign-Extended Displacements

Optimization

Use 8-bit sign-extended displacements for conditional branches.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using short, 8-bit sign-extended displacements for conditional branches improves code density with no negative affects on the processor.

4.11 Code Padding with Operand-Size Override and NOP

Optimization

Use one or more operand-size overrides (66h) and the NOP instruction (90h) to align code and space out branches.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Occasionally it is necessary to insert neutral code fillers into the code stream (for example, for code-alignment purposes or to space out branches). Because this filler code is executable, it should take up as few execution resources as possible, not diminish decode density, and not modify any processor state other than advancing the instruction pointer (rIP). Although there are several possible multibyte NOP-equivalent instructions that don't change the processor state (other than rIP), combinations of the operand-size override and the NOP instruction work best.

Example

Assign code-padding sequences like these and use them to align code and space out branches. These sequences are suitable for both 32-bit and 64-bit code, and you can use them on the AMD Athlon 64 and AMD Opteron processors, as well as seventh-generation AMD Athlon processors:

```
NOP1_OVERRIDE_NOP TEXTEQU <DB 090h>
NOP2_OVERRIDE_NOP TEXTEQU <DB 066h,090h>
NOP3_OVERRIDE_NOP TEXTEQU <DB 066h,066h,090h>
NOP4_OVERRIDE_NOP TEXTEQU <DB 066h,066h,066h,090h>
NOP5_OVERRIDE_NOP TEXTEQU <DB 066h,066h,090h,066h,090h>
NOP6_OVERRIDE_NOP TEXTEQU <DB 066h,066h,090h,066h,066h,090h>
NOP7_OVERRIDE_NOP TEXTEQU <DB 066h,066h,066h,090h,066h,066h,090h>
NOP8_OVERRIDE_NOP TEXTEQU <DB 066h,066h,066h,090h,066h,066h,066h,090h>
NOP9_OVERRIDE_NOP TEXTEQU <DB 066h,066h,090h,066h,066h,090h,066h,066h,090h>
```

For x87 floating-point instructions, a better single-byte padding exists. See "Align and Pack DirectPath x87 Instructions" on page 240.

Chapter 5 Cache and Memory Optimizations

The optimizations in this chapter take advantage of the large L1 caches and high-bandwidth buses of the AMD Athlon™ 64 and AMD Opteron™ processors.

This chapter covers the following topics:

5.1 Memory-Size Mismatches

Optimization

❖Avoid memory-size mismatches when different instructions operate on the same data. When one instruction stores and another instruction subsequently loads the same data, keep their operands aligned and keep the loads/stores of each operand the same size.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Examples—Store-to-Load-Forwarding Stalls

The following code examples result in a store-to-load-forwarding stall:

64-bit (Avoid)

```
foo DQ ?                    ; Assume foo is 8-byte aligned.
...
mov DWORD PTR foo, eax      ; Store a DWORD to foo.
mov DWORD PTR foo+4, ebx    ; Now store to foo+4.
mov rcx, QWORD PTR foo      ; Load a QWORD from foo.
```

32-bit (Avoid)

```
foo DQ ?                    ; Assume foo is 4-byte aligned.
...
mov DWORD PTR foo, eax      ; Store a DWORD in foo.
mov DWORD PTR foo+4, edx    ; Store a DWORD in foo+4.
fld QWORD PTR foo           ; Load a QWORD from foo.
```

Avoid

```
mov  foo, eax
mov  foo+4, edx
...
movq mm0, foo
```

Preferred

```
mov       foo, eax
mov       foo+4, edx
...
movd      mm0, foo
punpckldq mm0, foo+4
```

Preferred If Stores Are Close to the Load

```
movd      mm0, eax
mov       foo+4, edx
punpckldq mm0, foo+4
```

Examples—Large-to-small Mismatches

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

64-bit (Avoid)

```
foo DQ ?                    ; Assume foo is 8-byte aligned.
...
mov QWORD PTR foo, rax      ; Store a QWORD to foo.
mov eax, DWORD PTR foo      ; Load a DWORD from foo.
mov edx, DWORD PTR foo+4    ; Load a DWORD from foo+4.
```

32-bit (Avoid)

```
foo DQ ?                    ; Assume foo is 4-byte aligned.
...
fst QWORD PTR foo           ; Store a QWORD in foo.
mov eax, DWORD PTR foo      ; Load a DWORD from foo.
mov edx, DWORD PTR foo+4    ; Load a DWORD from foo+4.
```

Avoid

```
movq foo, mm0
...
mov  eax, foo
mov  edx, foo+4
```

Preferred

```
movd   foo, mm0
pswapd mm0, mm0
movd   foo+4, mm0
pswapd mm0, mm0
...
mov    eax, foo
mov    edx, foo+4
```

Preferred If the Contents of MM0 are No Longer Needed

```
movd      foo, mm0
punpckhdq mm0, mm0
movd      foo+4, mm0
...
mov       eax, foo
mov       edx, foo+4
```

Preferred If the Stores and Loads are Close Together, Option 1

```
movd   eax, mm0
pswapd mm0, mm0
movd   edx, mm0
pswapd mm0, mm0
```

Preferred If the Stores and Loads are Close Together, Option 2

```
movd      eax, mm0
punpckhdq mm0, mm0
movd      edx, mm0
```

5.2 Natural Alignment of Data Objects

Optimization

❖Make sure data objects are *naturally aligned*. An object is naturally aligned if it's located at an address that is a multiple of its size.

Locate this type of object	At an address evenly divisible by
Word	2
Doubleword	4
Quadword	8
Ten-byte (for example, TBYTE or REAL10)	8 (instead of 10)
Double quadword	16

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

A misaligned store or load operation suffers a minimum one-cycle penalty in the processor's load-store pipeline. Also, using misaligned loads and stores increases the likelihood of encountering a store-to-load forwarding pitfall, especially when operating in long mode (64-bit software). (For a more detailed discussion of store-to-load forwarding issues, see "Store-to-Load Forwarding Restrictions" on page 93.)

In addition, if the Alignment Mask bit is set in Control Register 0 (CR0), an unaligned memory reference may cause an alignment check exception. For more information on this topic, see Volume 2 of the *AMD64 Architecture Programmer's Manual* (order# 24593).

5.3 Multiprocessor Considerations

In a multiprocessor system, data within a single cache line that is shared between processors can reduce performance. In certain cases (for example, semaphores), this kind of cache-line data sharing cannot be avoided, but it should be minimized where possible.

Data can often be restructured so this does not occur. Cache lines on AMD Athlon 64 and AMD Opteron processors are presently 64 bytes, but a scheme that avoids this problem regardless of cache-line size makes for more performance-portable code.

For example, per-thread data can be allocated on the heap (for example, via calls to `malloc`), and this is preferred over statically defined shared arrays and variables that are potentially located in a single cache line. Furthermore, some software environments even provide special versions of `malloc` that guarantee data alignment to a specified value, and these can be useful in aligning data and eliminating unwanted cache line overlap.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

5.4 Store-to-Load Forwarding Restrictions

Store-to-load forwarding refers to the process of a load reading (forwarding) data from the store buffer. When this can occur, it improves performance because the load does not have to wait for the recently written (stored) data to be written to cache and then read back out again. There are instances in the load-store architecture of the AMD Athlon 64 and AMD Opteron processors when a load operation is not allowed to read needed data from a store in the store buffer.

In these cases, 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.

Due to the significant depth of the LS buffer of the AMD Athlon 64 and AMD Opteron processors, any load that is dependent on a store that cannot bypass data through the LS buffer may experience significant delays of up to tens of clock cycles, where the exact delay is a function of pipeline conditions.

The following sections describe store-to-load forwarding examples.

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 nor the 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. In these scenarios, a load has a true dependency on an 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 subrange of the addresses spanned by the load data.

Avoid

```
mov eax, 10h
mov WORD PTR [eax], bx     ; Word store
...
mov ecx, DWORD PTR [eax]   ; Doubleword load--can't forward upper byte
                           ;  from store buffer
```

Avoid

```
mov eax, 10h
mov BYTE PTR [eax+3], bl   ; Byte store
...
mov ecx, DWORD PTR [eax]   ; Doubleword load--can't 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 data.

Avoid

```
mov eax, 10h
add DWORD PTR [eax], ebx   ; Doubleword store
mov cx, WORD PTR [eax+2]   ; Word load--can't forward high word
                           ;  from store buffer
```

Avoid

```
movq [foo], mm1     ; Store upper and lower half.
...
add  eax, [foo]     ; Fine
add  edx, [foo+4]   ; Not good!
```

Preferred

```
movd      [foo], mm1     ; Store lower half.
punpckhdq mm1, mm1       ; Copy 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 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:

```
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 operation.
```

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:

```
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 AMD Athlon 64 and AMD Opteron processors. The lower 32 bits from an aligned quadword write feeding into a doubleword read is allowed, as illustrated in the following example:

```
movq [alignedQword], mm0
...
mov  eax, [alignedQword]
```

Store-to-Load Forwarding—False Dependencies

A load may detect a false dependency on a store-buffer entry if the load does not have a true dependency on the most recent store that matches address bits 11–2 of the load. A false match could occur on the most recent store that writes somewhere within the same doubleword of memory as the load. In addition, a false match could occur if a store address is located at an exact multiple of

4-Kbyte pages away from the load address (address bits 47–12 do not match). Avoid the type of code shown in the following example:

```
mov eax, 10h
mov [eax], bx     ; Word store to address 10
mov cx, [eax+2]   ; Word load to address 12
                  ; Load detects a false dependency
                  ;  on store because it is in the
                  ;  same doubleword of memory.
mov cx, [eax+4]   ; Word load to address 14
                  ; Load does not detect a false
                  ;  dependency because it is to a
                  ;  different doubleword of memory.
```

Here's another example of the type of code to avoid:

```
mov eax, 10h
mov [eax], bl     ; First store to DWORD at address 10h
mov [eax+1], cl   ; Second store to DWORD at address 10h
mov dl, [eax]     ; Load detects a false
                  ;  dependency on the second store
                  ;  because it is the most recent
                  ;  store to the same doubleword of
                  ;  memory as the load.
```

Summary of Store-to-Load-Forwarding Pitfalls to Avoid

To avoid store-to-load-forwarding pitfalls, follow these 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.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

5.5 Prefetch Instructions

Optimization

Where appropriate, use one of the prefetch instructions to increase the effective bandwidth of the AMD Athlon 64 and AMD Opteron processors.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Prefetch instructions take advantage of the high bus bandwidth of the AMD Athlon 64 and AMD Opteron processors to hide latencies when fetching data from system memory. A prefetch instruction initiates a read request of a specified address and reads the entire cache line that contains that address.

AMD Athlon 64 and AMD Opteron processors perform three types of prefetches:

Prefetch type	Description
Load	Reads the data into the L1 data cache; the data is later evicted to the L2 cache. The following instructions perform load prefetches: PREFETCH, PREFETCHT0, PREFETCHT1, and PREFETCHT2.
Store	Reads the data into the L1 data cache and marks the data as modified; the data is later evicted to the L2 cache. The PREFETCHW instruction performs a store prefetch.
Nontemporal	Reads the data into way 0 of the L1 data cache; the data is never evicted to the L2 cache, avoiding cache pollution. The PREFETCHNTA instruction performs a nontemporal prefetch.

The prefetch instructions can be used anywhere, in any type of code. The use of prefetch instructions is not affected by the values of Control Register 0 (CR0) bits, such as CR0.EM and CR0.TS.

Prefetching versus Preloading

In code that makes irregular memory accesses rather than sequential accesses, an ordinary MOV instruction is the best way to load data. But in situations where sequential addresses are read, prefetch instructions can improve performance. Prefetch instructions only update the L1 data cache and do not update an architectural register. This uses one less register compared to a load instruction.

Unit-Stride Access

Large data sets typically require unit-stride access to ensure that all data pulled in by a prefetch instruction is actually used. Large data sets make use of all data that is read from memory, rather than using only a sparse subset of the memory. If necessary, you should reorganize algorithms or data structures to allow unit-stride access. For a definition of unit-stride access, see “Definitions” on page 103.

Hardware Prefetching

The AMD Athlon 64 and AMD Opteron processors implement a hardware prefetching mechanism. The prefetched data is loaded into the L2 cache. The hardware prefetcher works most efficiently when data is accessed on a cache-line-by-cache-line basis (that is, without skipping cache lines). Cache lines on current AMD Athlon 64 and AMD Opteron processors are 64 bytes, but cache-line size is implementation dependent.

In some cases, using prefetch instructions on processors with hardware prefetching may slightly reduce performance. In these cases, it may be necessary to remove the prefetch instructions. You should weigh the measured gains obtained on non-hardware-prefetch-enabled processors using the software prefetch instruction against any loss in performance on processors with the hardware prefetcher. All current AMD Athlon 64 and AMD Opteron processors have hardware prefetching mechanisms.

The hardware prefetcher prefetches data that is accessed in an ascending order on a cache-line-by-cache-line basis. When the hardware prefetcher detects an access to cache line *l* followed by an access to cache line *l* + 1, it initiates a prefetch of cache line *l* + 3. Accessing data in increments larger than 64 bytes may fail to trigger the hardware prefetcher because cache lines are skipped. In these cases, software-prefetch instructions should be employed. The hardware prefetcher also is not triggered when code accesses memory in a descending order.

PREFETCH/W versus PREFETCHNTA/T0/T1/T2

PREFETCHNTA, PREFETCHT0, PREFETCHT1, and PREFETCHT2 are SSE instructions and are processor-implementation dependent. For the AMD Athlon 64 and AMD Opteron processors, data that is prefetched with the PREFETCHNTA instruction is not placed into the L2 cache when it is evicted. This instruction is intended for non-temporal data that will not be needed again soon. Chapters 5 and 9 show examples of how to use the PREFETCHNTA instruction.

Current AMD Athlon 64 and AMD Opteron processors implement the PREFETCHT0, PREFETCHT1 and PREFETCHT2 instructions in exactly the same way as the PREFETCH instructions. That is, the data is brought into the L1 data cache. This functionality could be changed in future implementations.

PREFETCHW versus PREFETCH

Code that intends to modify the cache line that is brought in through prefetching should use the PREFETCHW instruction. PREFETCHW gives a hint to the AMD Athlon 64 and AMD Opteron

processors of an intent to modify the cache line. The AMD Athlon 64 and AMD Opteron processors mark the cache line being read by PREFETCHW as *modified*. Using PREFETCHW can save additional cycles compared to PREFETCH, and avoid the subsequent cache state change caused by a write to the prefetched cache line. Only use PREFETCHW if there is a write to the same cache line afterwards.

Write-Combining Usage

Use write-combining instructions instead of PREFETCHW in situations where all of the following conditions are true:

- The code will overwrite one or more complete cache lines with new data.
- The new data will not be used again soon.

Write-combining instructions include the SSE and SSE2 instructions MOVNTDQ, MOVNTI, MOVNTPS, and MOVNTPD. They also include the MMX instruction MOVNTQ.

Write-combining instructions can dramatically improve memory-write performance. They write data directly to memory through write-combining buffers, bypassing the cache. This is faster than PREFETCHW because data doesn't need to be initially read from memory to fill the cache lines, only to be completely overwritten shortly thereafter. The new data is simply written to memory, replacing the old data in memory, so no memory read is performed.

One application where write-combining is useful, often in conjunction with prefetch instructions, is in copying large blocks of memory.

Note: *The write-combining instructions are not recommended or necessary for write-combined memory regions as the processor will automatically combine writes for those regions. Write-combine memory types are indicated through the MTRRs and the page-attribute table (PAT).*

For more information on write-combining, see Appendix B, "Implementation of Write-Combining."

Multiple Prefetches

Programmers can initiate multiple outstanding prefetches on the AMD Athlon 64 and AMD Opteron processors. The AMD Athlon 64 and AMD Opteron processors can have a theoretical maximum of eight outstanding prefetches, but in practice the number is usually smaller. When all resources are filled by various memory read requests, the processor waits until resources become free before processing the next request. Multiple prefetch requests are essentially handled in order, prefetching data in the order that it is needed.

The following example shows how to initiate multiple prefetches when traversing more than one array.

Example—Multiple Prefetches

```
.CODE
.K3D
.686
```

```
; Original C code:
;
; #define LARGE_NUM 65536
; #define ARR_SIZE (LARGE_NUM*8)
;
; 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 edx, (-LARGE_NUM)     ; Use biased index.
   mov eax, OFFSET array_a   ; Get address of array_a.
   mov ebx, OFFSET array_b   ; Get address of array_b.
   mov ecx, OFFSET array_c   ; Get address of array_c.

loop:
   prefetchw [eax+256]   ; Four cache lines ahead
   prefetch  [ebx+256]   ; Four cache lines ahead
   prefetch  [ecx+256]   ; Four cache lines ahead
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE]       ; b[i]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE]       ; b[i] * c[i]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE]       ; a[i] = b[i] * c[i]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+8]     ; b[i+1]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+8]     ; b[i+1] * c[i+1]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+8]     ; a[i+1] = b[i+1] * c[i+1]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+16]    ; b[i+2]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+16]    ; b[i+2]*c[i+2]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+16]    ; a[i+2] = [i+2] * c[i+2]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+24]    ; b[i+3]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+24]    ; b[i+3] * c[i+3]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+24]    ; a[i+3] = b[i+3] * c[i+3]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+32]    ; b[i+4]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+32]    ; b[i+4] * c[i+4]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+32]    ; a[i+4] = b[i+4] * c[i+4]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+40]    ; b[i+5]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+40]    ; b[i+5] * c[i+5]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+40]    ; a[i+5] = b[i+5] * c[i+5]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+48]    ; b[i+6]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+48]    ; b[i+6] * c[i+6]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+48]    ; a[i+6] = b[i+6] * c[i+6]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+56]    ; b[i+7]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+56]    ; b[i+7] * c[i+7]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+56]    ; a[i+7] = b[i+7] * c[i+7]
   add  edx, 8   ; Compute next 8 products
   jnz  loop     ;  until none left.
```

```
END
```

The following optimization rules are applied to this example:

- Partially unroll loops to ensure that the data stride per loop iteration is equal to the length of a cache line. This avoids overlapping PREFETCH instructions and thus makes optimal use of the available number of outstanding prefetches.
- Because the array `array_a` is written rather than read, use PREFETCHW instead of PREFETCH to avoid overhead for switching cache lines to the correct state. The prefetch distance is optimized such that each loop iteration is working on three cache lines while active prefetches bring in the next cache lines.
- Reduce index arithmetic 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

When determining how far ahead to prefetch, the basic guideline is to initiate the prefetch early enough so that the data is in the cache by the time it is needed, under the constraint that there can't be more than eight prefetches in flight at any given time.

To determine the optimal prefetch distance, use empirical benchmarking when possible. Prefetching three or four cache lines ahead (192 or 256 bytes) is a good starting point and usually gives good results. Trying to prefetch too far ahead impairs performance.

Memory-Limited versus Processor-Limited Code

Software prefetching can help to hide the memory latency, but it can't increase the total memory bandwidth. Many loops are limited by memory bandwidth rather than processor speed, as shown in Figure 1. In these cases, the best that software prefetching can do is to ensure that enough memory requests are "in flight" to keep the memory system busy all of the time. The AMD Athlon 64 and AMD Opteron processors support a maximum of eight concurrent memory requests to different cache lines. Multiple requests to the same cache line count as only one towards this limit of eight

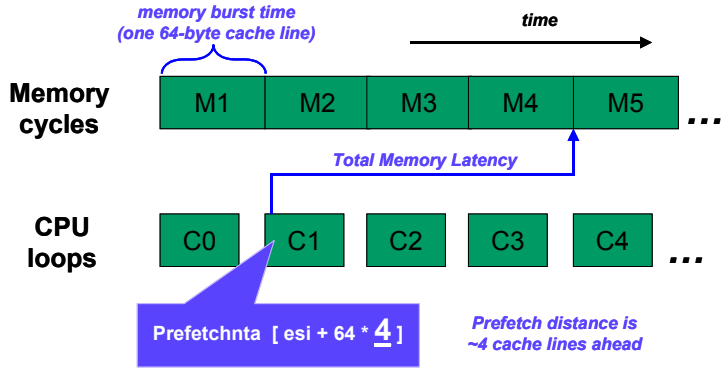


Figure 1. Memory-Limited Code

Code that performs many computations on each cache line is limited by processor speed rather than memory bandwidth, as shown in Figure 2. In this case, the goal of software prefetching is just to ensure that the memory data is available when the processor needs it. As the processor speed increases, the optimal prefetch distance increases until the memory bandwidth becomes the limiting factor.

For an example of how to use software prefetching in processor-limited code, see “Structuring Code with Prefetch Instructions to Hide Memory Latency” on page 198.

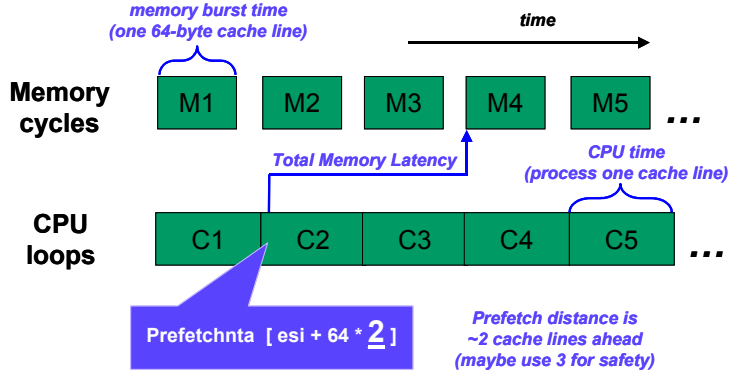


Figure 2. Processor-Limited Code

Definitions

Unit-stride access refers to a memory access pattern where consecutive memory accesses are made to consecutive array elements, in ascending or descending order. If the arrays are made of elemental types, then they imply adjacent memory locations as well. For example:

```
char j, k[MAX];
for (i = 0; i < MAX; i++) {
...
j += k[i];   // Every byte is used.
...
}
double x, y[MAX];
for (i = 0; i < MAX; i++) {
...
x += y[i];   // Every byte is used.
...
}
```

Exception to Unit Stride

The unit-stride concept works well when stepping through arrays of elementary data types. In some instances, unit stride alone may not be sufficient to determine how to use the PREFETCH instruction properly. For example, assume that there is a vertex structure of 256 bytes and the code steps through the vertices in unit stride, but using only the x, y, z, w components, each being of type `float` (for example, the first 16 bytes of each vertex). In this case, the prefetch distance obviously should be some function of the data size structure (for a properly chosen *n*):

```
prefetch [eax+n*structure_size]
...
add      eax, structure_size
```

You should experiment to find the optimal prefetch distance; there is no formula that works for all situations.

Data Stride per Loop Iteration

Assuming unit-stride access to a single array, the data stride of a loop (the *loop stride*) refers to the number of bytes accessed in the array per loop iteration. For example:

```
   fldz
add_loop:
   fadd QWORD PTR [ebx*8+base_address]
   dec  ebx
   jnz  add_loop
```

The data stride of the above loop is eight bytes. In general, for optimal use of prefetching, the data stride per iteration is the length of a cache line (64 bytes in the AMD Athlon 64 and AMD Opteron processors). If the loop stride is smaller, unroll the loop enough to use a whole cache line of data per

iteration. However, unrolling the loop may not be feasible if the original loop stride is very small (for example, only two bytes).

Prefetch at Least 64 Bytes Away from Surrounding Stores

The prefetch 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 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.

5.6 Write-combining

Optimization

❖Operating-system, device-driver, and BIOS programmers should take advantage of the write-combining capabilities of the AMD Athlon 64 and AMD Opteron processors.

For details, see Appendix B, “Implementation of Write-Combining.” For more information on write-combining, see “Write-Combining” in Volume 2 of the *AMD64 Architecture Programmer’s Manual* (order no. 24593).

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In order to improve system performance, the AMD Athlon 64 and AMD Opteron processors aggressively combine multiple memory-write cycles (of any data size) that address locations within a 64-byte cache-line-aligned write buffer.

5.7 L1 Data Cache Bank Conflicts

Optimization

Utilize pair loads that do not have a bank conflict in the L1 data cache to improve load thoughput.

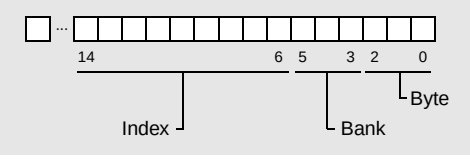
Application

This optimization applies to:

- 32-bit software
- 64-bit software

Fields Used to Address the Multibank L1 Data Cache

The L1 data cache is a multibank design consisting of 8 banks total, where each bank is 8 bytes wide. To address the L1 data cache, the processor uses fields within the address as shown in the following diagram:



How to Know If a Bank Conflict Exists

The existence of a bank conflict between two neighboring loads depends on their bank and index values:

When the bank is	And the index is	Then a bank conflict
Different	Either the same or different	Doesn't exist
The same	The same	Doesn't exist
The same	Different	Exists

In other words, with common data types, consecutive array elements can't have a bank conflict. If the array elements are 4 bytes or less, the two loads are to the same index and the same bank, and no conflict occurs. If the array elements are 8 bytes, the loads are to the same index but different banks, so a bank conflict doesn't occur either.

Rationale

Loads are served by the L1 data cache in program order, but the number of loads that the processor can perform in one cycle depends on whether a bank conflict exists between the loads:

When a bank conflict	Then the number of loads the processor can perform per cycle is
Exists	1
Doesn't exist	2

Therefore, pairing loads that don't have a bank conflict helps maximize load throughput.

Example

Avoid code like this, where two loads without a bank conflict are separated by other instructions:

```
fld   qword ptr [eax]
fmul  qword ptr [ebx]
faddp st(3), st
fld   qword ptr [eax+8]
fmul  qword ptr [ebx+8]
faddp st(2), st
```

Instead, rearrange the two loads so they appear as a pair:

```
fld   qword ptr [eax]
fld   qword ptr [eax+8]
fmul  qword ptr [ebx+8]
faddp st(2), st
fmul  qword ptr [ebx]
faddp st(3), st
```

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

Optimization

❖Avoid placing code and data together within a cache line, especially if the data becomes modified.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Sharing code and data in the same 64-byte cache line may cause the L1 caches to thrash (unnecessarily cast out code or data) in order to maintain coherency between the separate instruction and data caches. The AMD Athlon 64 and AMD Opteron processors have a cache-line size of 64 bytes.

For example, 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. This mixing of code and data in the same cache line results in lower performance.

Don’t place critical code at the border between 32-byte-aligned code segments and data segments. Code at the beginning or end of a data segment should be executed as infrequently as possible or padded.

In summary, avoid self-modifying code and storing data in code segments.

5.9 Sorting and Padding C and C++ Structures

Optimization

Sort and pad C and C++ structures to achieve natural alignment.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

By sorting and padding structures at the source-code level, if the first member of a structure is naturally aligned, then all other members are naturally aligned as well. This allows, for example, arrays of structures to be perfectly aligned.

Sorting and Padding C and C++ Structures

To sort and pad a C or C++ structure, follow these steps:

1. Sort the structure members according to their type sizes, declaring members with larger type sizes ahead of members with smaller type sizes.
2. Pad the structure so the size of the structure is a multiple of the largest member's type size.

Example

Consider the following structure declaration in a C function:

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

Instead of allocating the members in the order in which they're declared, allocate them from lower to higher addresses in the following order and add padding:

`x, k, a[4], a[3], a[2], a[1], a[0],` *`pad_byte6`*`,..., `*`pad_byte0`*

Related Information

For information on sorting and padding C and C++ structures at the C-source level, see “Sorting and Padding C and C++ Structures” on page 39.

5.10 Sorting Local Variables

Optimization

Sort local variables according to their type sizes, allocating those with larger type sizes ahead of those with smaller type sizes.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

If the first variable is allocated for natural alignment, all other variables are allocated contiguously in the order they are declared and are naturally aligned without any padding.

Example

Consider the following declarations in a C function:

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

Instead of allocating the variables in the order in which they're declared, allocate them from lower to higher addresses in the following order:

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

Related Information

For information on sorting local variables at the C-source level, see "Sorting Local Variables" on page 41.

5.11 Appropriate Memory Copying Routines

Optimization

❖Use the appropriate routine when copying a block of memory. This section provides examples of routines you can use to copy blocks of memory and explains how to improve performance by choosing the appropriate routine for a given situation.

Application

This optimization applies to:

- 64-bit software

Types of Routines as Classified by Block Size

With regard to block size, there are two types of block-copy routines:

Routine type	Description
Small block	Designed for use with small blocks (blocks that are the same size as or smaller than the L1 data cache) and when the data in the block will be needed again soon. Small block-copy routines leave the destination data in the L1 data cache for fast access by subsequent instructions. The performance numbers given for small block copies were measured with the source data in the L1 data cache and the cache lines already allocated for the destination data. Because these numbers are entirely dependent on the speed of the processor, they are given in units of bytes/clock.
Large block	Designed for use with large blocks (blocks that are larger than the L1 data cache) or when the data in the block won't be needed again soon. Large block-copy routines use prefetch instructions to efficiently read the source data from main memory; these routines also use write-combining instructions to efficiently write destination data directly to main memory without polluting any of the caches. The performance numbers given for large block copies (in the comments of the routines that follow) were measured using single-channel DDR333 memory with CAS2 timing. For comparison purposes, performance numbers are given for both a small (8-Kbyte) block and a large (8-Mbyte) block. The 8-Kbyte numbers are provided for comparison purposes only. The performance numbers given for 8-Kbyte blocks were measured with the source data in the L1 data cache, so these numbers are higher than for 8-Mbyte blocks, where the source data is not in the cache. The measurements for both 8-Kbyte and 8-Mbyte block copies would be the same if the 8-Kbyte block was read from main memory instead of the L1 data cache.

Note: *All performance numbers count total traffic, which includes both a read and a write for each byte copied.*

Types of Alignment

There are two basic types of alignment for a block of memory or data object:

Alignment type	Description
Aligned	A block of memory or a data object whose starting address is evenly divisible by a particular power of two. For example, an aligned block might be 8-byte aligned.
Unaligned	A block of memory or a data object whose starting address is not evenly divisible by a power of two.

Granularity

The *granularity* of a routine is the number of bytes that the routine's outer loop copies at a time and, therefore, the minimum number of bytes that the routine can copy.

Determining Which Routine to Use

In terms of performance, the best routine to use to copy a block of memory depends on the size and alignment of the block.

Use Table 6 to determine which routine to use for a given block.

Table 6. Routine Selection for Block Copies

When the block to be copied is	Then use the routine in this section	On page
Less than or equal to 64 bytes	Routine 3: Small Aligned Block Copy Using MOVSQ	118
• Greater than 64 bytes • Small enough to fit in the L1 data cache • Needed again soon	Routine 4: Small Block Copy Using Discrete Moves	119
• Greater than 64 bytes • Small enough to fit in the L1 data cache • *Not* needed again soon	Routine 6: Large Aligned Block Copy Using Discrete Moves (Fast)	121
• Greater than 64 bytes • Larger than the L1 data cache • Needed again soon	Routine 6: Large Aligned Block Copy Using Discrete Moves (Fast)*	121
• Greater than 64 bytes • Larger than the L1 data cache • *Not* needed again soon	Routine 6: Large Aligned Block Copy Using Discrete Moves (Fast)	121

Note:

** For blocks larger than the L1 data cache but smaller than the L2 cache, the small copy routines may provide better performance.*

Aligning the Destination Starting Address

If the original destination address doesn't meet the recommended routine's alignment requirements, follow these steps:

1. Copy the bytes whose destination addresses are smaller than the lowest properly aligned destination address using the routine in "Routine 1: Small, General Block Copy Using MOVSB" on page 116.
2. Use the recommended routine to copy the remainder of the block.

Copying Remaining Bytes

If uncopied bytes remain after using the recommended routine, do this:

- Copy the remaining bytes using the routine in “Routine 1: Small, General Block Copy Using MOVSB” on page 116.

Routine 1: Small, General Block Copy Using MOVSB

This routine illustrates a fast and simple way to copy a block of memory. You can use this routine to copy any block, regardless of its alignment; however, this routine is not an efficient way to copy a block of more than about 64 bytes.

Reserve this routine for copying unaligned or two-byte-aligned blocks of memory that are less than 64 bytes.

```
; Throughput (8-Kbyte block): ~1.4 bytes/clock
; Throughput (8-Mbyte block): ~0.79 Gbytes/s (The performance for large blocks
;  is shown for comparison purposes only. This routine is not recommended
;  for large blocks.)

; Destination alignment: Any
; Source alignment: Any
; Granularity: 1 byte
; Input:
;  RCX = destination address (dest)
;  RDX = source address (src)
;  R8 = number of bytes to copy (count)

mc_movsb PROC NEAR
   ; Save RSI and RDI.
   ...
   mov rsi, rdx   ; Load source address.
   mov rdi, rcx   ; Load destination address.
   mov rcx, r8    ; Load number of bytes to copy.
   rep movsb      ; Copy bytes until none remain.
   ...
   ; Restore RSI and RDI.
   ret
mc_movsb ENDP
```

Routine 2: Small Aligned Block Copy Using MOVSD

This routine shows a fast and simple way to copy a block of data that is 4-byte aligned, not 8-byte aligned.

```
; Throughput (8-Kbyte block): ~5-6 bytes/clock
; Throughput (8-Mbyte block): ~1.05 Gbytes/s (The performance for large blocks
;  is shown for comparison purposes only. This routine is not recommended
;  for large blocks.)

; Destination alignment: none required
; Source alignment: none required
; Granularity: 4 bytes
; Input:
;  RCX = destination address (dest)
;  RDX = source address (src)
;  R8 = number of bytes to copy (count)

mc_movsd PROC NEAR
   ; Save RSI and RDI.
   ...
   mov rsi, rdx   ; Load source address.
   mov rdi, rcx   ; Load destination address.
   mov rcx, r8    ; Load number of bytes to copy.
   shr rcx, 2     ; Convert number of bytes to number of DWORDs.
   rep movsd      ; Copy DWORDs until none remain.
   ...
   ; Restore RSI and RDI.
   ret
mc_movsd ENDP
```

Routine 3: Small Aligned Block Copy Using MOVSQ

This routine shows a fast and simple way to copy a block of data that is 8-byte aligned.

```
; Throughput (8-Kbyte block): ~11-12 bytes/clock
; Throughput (8-Mbyte block): 1.22 Gbytes/s (The performance for large blocks
;  is shown for comparison purposes only. This routine is not recommended
;  for large blocks.)

; Destination alignment: none required
; Source alignment: none required
; Granularity: 8 bytes
; Input:
;  RCX = destination address (dest)
;  RDX = source address (src)
;  R8 = number of bytes to copy (count)

mc_movsq PROC NEAR
   ; Save RSI and RDI.
   ...
   mov rsi, rdx   ; Load source address.
   mov rdi, rcx   ; Load destination address.
   mov rcx, r8    ; Load number of bytes to copy.
   shr rcx, 3     ; Convert number of bytes to number of QWORDs.
   rep movsq      ; Copy QWORDs until none remain.
   ...
   ; Restore RSI and RDI.
   ret
mc_movsq ENDP
```

Routine 4: Small Block Copy Using Discrete Moves

When compared with "Routine 3: Small Aligned Block Copy Using MOVSQ" on page 118, this routine provides better performance for a block of memory that is 8-byte aligned. Although the size of the code is larger, the performance is 10–15% higher.

```
; Throughput (8-Kbyte block): ~11-14 bytes/clock

; Destination alignment: none required
; Source alignment: none required
; Granularity (sub-block size): 32 bytes
; Input:
;  RCX = destination address (dest)
;  RDX = source address (src)
;  R8 = number of bytes to copy (count)

mc_sm_aligned PROC NEAR
   add rdx, r8   ; Add to source address the number of bytes to copy.
   add rcx, r8   ; Add to destination address the number of bytes to copy.
   shr r8, 3     ; Convert number of bytes to number of QWORDs.
   neg r8        ; Make number of QWORDs negative.
   ALIGN 16      ; Pad to align top of loop.
copyloop:
   mov rax, [rdx+r8*8]      ; Copy sub-block (four QWORDs).
   mov [rcx+r8*8], rax
   mov rax, [rdx+r8*8+8]
   mov [rcx+r8*8+8], rax
   mov rax, [rdx+r8*8+16]
   mov [rcx+r8*8+16], rax
   mov rax, [rdx+r8*8+24]
   mov [rcx+r8*8+24], rax
   add r8, 4                ; Decrement number of QWORDs to copy.
   jnz copyloop             ; If QWORDs remain, then jump.
   ret
mc_sm_aligned ENDP
```

Routine 5: Large, Aligned Block Copy Using Discrete Moves (Basic)

This routine is a easy way to copy a large block of memory; however, if the block is larger than eight Kbytes, you can achieve better performance using the routine in "Routine 6: Large Aligned Block Copy Using Discrete Moves (Fast)" on page 121.

This routine copies a 16-byte-aligned block of memory to a 16-byte-aligned destination address using a relatively small granularity of 32 bytes.

If the original destination address is not 16-byte aligned, you can achieve the required 16-byte destination alignment by separately copying the bytes that precede the lowest 16-byte-aligned address in the destination block. For more information, see "Aligning the Destination Starting Address" on page 114.

```
; Throughput (8-Kbyte block): ~4.7 Gbytes/s (The performance for small blocks
;  is shown for comparison purposes only.)
; Throughput (8-Mbyte block): ~2.0 Gbytes/s

; Destination alignment: 16 bytes
; Source alignment: 16 bytes
; Granularity (sub-block size): 32 bytes
; Input:
;  RCX = destination address (dest)
;  RDX = source address (src)
;  R8 = number of bytes to copy (count)

mc_lg_aligned_basic PROC NEAR
   add rdx, r8   ; Add to source address the number of bytes to copy.
   add rcx, r8   ; Add to destination address the number of bytes to copy.
   shr r8, 3     ; Convert number of bytes to number of QWORDs.
   neg r8        ; Make number of QWORDs negative.

copyloop:
   prefetchnta [rdx+r8*8+256]    ; Load a line into the L1 data cache; mark the
                                 ;  line so it won't be evicted to L2.
   movdqa  xmm0, [rdx+r8*8]      ; Copy sub-block (four QWORDs) using
   movntdq  [rcx+r8*8], xmm0     ; write-combining buffer.
   movdqa xmm1, [rdx+r8*8+16]
   movntdq [rcx+r8*8+16], xmm1
   add r8, 4                     ; Decrement number of QWORDs to copy.
   jnz copyloop                  ; If QWORDs remain, then jump.
   sfence                        ; Flush the write-combining buffer.
   ret
mc_lg_aligned_basic ENDP
```

Routine 6: Large Aligned Block Copy Using Discrete Moves (Fast)

This routine provides excellent performance for large block sizes. It copies a 16-byte-aligned block of memory to a 16-byte-aligned destination address using a relatively large granularity of eight Kbytes. To copy a block of memory smaller than eight Kbytes, see “Routine 5: Large, Aligned Block Copy Using Discrete Moves (Basic)” on page 120.

This routine avoids frequent alternation between reads and writes by first loading eight Kbytes of data into the L1 data cache without interruption, and then writing that chunk of data to the destination address in main memory. If necessary, you can modify this routine to copy data in smaller chunks; however, reducing the granularity also reduces the performance.

If the original destination address is not 16-byte aligned, you can achieve the required 16-byte destination alignment by separately copying the bytes that precede the lowest 16-byte-aligned address in the destination block. For more information, see “Aligning the Destination Starting Address” on page 114.

```
; Throughput (8-Kbyte block): ~4.7 Gbytes/s (The performance for small blocks
;  is shown for comparison purposes only.)
; Throughput (8-Mbyte block): ~2.3 Gbytes/s

; Destination alignment: 16 bytes
; Source alignment: 16 bytes
; Granularity (chunk size): 8 Kbytes
; Input:
;  RCX = destination address (dest)
;  RDX = source address (src)
;  R8 = number of bytes to copy (count)

mc_lg_aligned_fast PROC NEAR
   add rdx, r8   ; Add to source address the number of bytes to copy.
   add rcx, r8   ; Add to destination address the number of bytes to copy.
   shr r8, 3     ; Convert number of bytes to number of QWORDs.
   neg r8        ; Make number of QWORDs negative.

chunkloop:
   ; Prefetch and copy a chunk (8 Kbytes) of memory.
   mov r9, r8               ; Save number of QWORDs.
   mov r10, 64              ; Initialize number of cache-line pairs to prefetch.

prefetchloop:
   prefetchnta [rdx+r8*8]      ; Load a line (64 bytes) into the L1 data cache;
                               ;  mark the line so it won't be evicted to L2.
   prefetchnta [rdx+r8*8+64]   ; Load next cache line.
   add r8, 16                  ; Select next cache-line pair.
   dec r10                     ; Decrement number of cache-line pairs.
   jnz prefetchloop            ; If cache-line pairs remain, then jump.
   mov r8, r9                  ; Restore number of QWORDs.

   mov eax, 64*4                 ; Initialize number of sub-blocks to copy.
```

```
copyloop:
   movdqa  xmm0, [rdx+r8*8]      ; Copy sub-block (four QWORDs) using
   movntdq [rcx+r8*8], xmm0      ; write-combining buffer.
   movdqa xmm1, [rdx+r8*8+16]
   movntdq [rcx+r8*8+16], xmm1
   add r8, 4                     ; Select next sub-block.
   dec eax                       ; Decrement number of sub-blocks to copy.
   jnz copyloop                  ; If another sub-block remains, then jump.
   or  r8, r8                    ; Test whether chunk count is 0.
   jnz chunkloop                 ; If another chunk remains, then jump.
   sfence                        ; Flush the write-combining buffer.
   ret
mc_lg_aligned_fast ENDP
```

5.12 Stack Considerations

Make sure the stack is suitably aligned for the local variable with the largest base type. Then, using the technique described in “Sorting and Padding C and C++ Structures” on page 109, all variables can be properly aligned with no padding.

Application

This optimization applies to:

- 32-bit software

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

```
prologue:
   push ebp
   mov  ebp, esp
   sub  esp, SIZE_OF_LOCALS   ; Size of local variables
   and  esp, –8
   ...                        ; Push registers that need to be preserved.

epilogue:                     ; Pop register that needed to be preserved.
   leave
   ret
```

With this technique, function arguments can be accessed through EBP, and local variables can be accessed through ESP. Save and restore EBP between the prologue and the epilogue to keep it free for general use.

Optimized Stack Usage

It is sometimes possible to improve performance in frequently executed routines by altering the way variables and parameters are passed and accessed on the stack. Replacing PUSH and POP instructions with MOV instructions can reduce stack pointer dependencies and uses fewer execution resources. This optimization is usually most effective in smaller routines. Excessive use of this optimization can result in increased code size as MOV instructions are considerably larger than PUSH and POP instructions.

5.13 Interleave Loads and Stores

When loading and storing data as in a copy routine, the organization of the sequence of loads and stores can affect performance.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When using SSE and SSE2 instructions to perform loads and stores, it is best to interleave them in the following pattern—Load, Store, Load, Store, Load, Store, etc. This enables the processor to maximize the load/store bandwidth.

If using MMX loads and stores in 32-bit mode, the loads and stores should be arranged in the following pattern—Load, Load, Store, Store, Load, Load, Store, Store, etc.

Example

The following example illustrates a sequence of 128-bit loads and stores:

```
movdqa     xmm0,[rdx+r8*8]          ; Load
movntdq    [rcx+r8*8],xmm0          ; Store
movdqa      xmm1,[rdx+r8*8+16]      ; Load
movntdq    [rcx+r8*8+16],xmm1       ; Store
```

Chapter 6 Branch Optimizations

The optimizations in this chapter help improve branch prediction and minimize branch penalties.

In This Chapter

This chapter covers the following topics:

6.1 Density of Branches

Optimization

When possible, align branches such that they don’t cross a 16-byte boundary.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The AMD Athlon™ 64 and AMD Opteron™ processors have the capability to cache branch-prediction history for a maximum of three near branches (CALL, JMP, conditional branches, or returns) per 16-byte fetch window. A branch instruction that crosses a 16-byte boundary is counted in the second 16-byte window. Due to architectural restrictions, a branch that is split across a 16-byte boundary cannot dispatch with any other instructions when it is predicted taken. Perform this alignment by rearranging code; it’s not beneficial to align branches using padding sequences.

The following branches are limited to three per 16-byte window:

```
jcc  rel8
jcc  rel32
jmp  rel8
jmp  rel32
jmp  reg
jmp  WORD PTR
jmp  DWORD PTR
call rel16
call r/m16
call rel32
call r/m32
```

Coding more than three branches in the same 16-byte code window may lead to conflicts in the branch target buffer. To avoid conflicts in the branch target buffer, space out branches such that three

or fewer exist in a given 16-byte code window. For absolute optimal performance, try to limit branches to one per 16-byte code window. Avoid code sequences like the following:

```
ALIGN 16

label3:
   call label1   ; 1st branch in 16-byte code window
   jc   label3   ; 2nd branch in 16-byte code window
   call label2   ; 3rd branch in 16-byte code window
   jnz  label4   ; 4th branch in 16-byte code window
                 ; Cannot be predicted.
```

If there is a jump table that contains many frequently executed branches, pad the table entries to 8 bytes each to assure that there are never more than three branches per 16-byte block of code.

Only branches that have been taken at least once are entered into the dynamic branch prediction, and therefore only those branches count toward the three-branch limit.

6.2 Two-Byte Near-Return RET Instruction

Optimization

Use of a two-byte near-return can improve performance. The single-byte near-return (opcode C3h) of the RET instruction should be used carefully. Specifically, avoid the following two situations:

- Any kind of branch (either conditional or unconditional) that has the single-byte near-return RET instruction as its target. See “Examples.”
- A conditional branch that occurs in the code directly before the single-byte near-return RET instruction. See “Examples.”

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The processor is unable to apply a branch prediction to the single-byte near-return form (opcode C3h) of the RET instruction.

The easiest way to assure the utilization of the branch prediction mechanism is to use a two-byte RET instruction. A two-byte RET has a REP instruction inserted before the RET, which produces the functional equivalent of the single-byte near-return RET instruction, but is not affected by the prediction limitations outlined above. To use a two-byte RET, define a text macro named `REPRET` and use it instead of the RET instruction to force the intended object code.

```
REPRET TEXTEQU <DB 0F3h, 0C3h>
```

Examples

Avoid branches in which the target of the branch is a single-byte near-return:

```
   jmp label   ; Jump to a single-byte near-return RET instruction.
   ...
label:
   ret         ; RET is potentially mispredicted.
```

Avoid branches that immediately precede a single-byte near-return:

```
jz  label   ; Conditional branch is not taken.
ret         ; RET is a fall-through instruction,
            ;  potentially mispredicted.
```

If possible, move an existing instruction, such as a POP instruction that is part of the function epilogue, so that it's inserted between the branch and the RET instruction:

```
jz  label
pop ebp   ; Pad with at least one non-branch instruction.
ret
```

If no existing instruction is available for this purpose, then insert a NOP instruction to provide the necessary padding or, better still, use the recommended two-byte version of RET.

6.3 Branches That Depend on Random Data

Optimization

Avoid conditional branches that depend on random data, as these branches are difficult to predict.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Suppose 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 cause the branch-prediction logic to mispredict the branch about 50% of the time.

If possible, design branch-free alternative code sequences that result in shorter average execution time. This technique is especially important if the branch body is small.

Examples

The following examples illustrate this concept using the CMOV*xx* instruction.

Signed Integer ABS Function (x = labs(x))

```
mov   ecx, [x]   ; Load value.
mov   ebx, ecx   ; Save value.
neg   ecx        ; Negate value.
cmovs ecx, ebx   ; If negated value is negative, select value.
mov   [x], ecx   ; Save labs result.
```

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).
```

Conditional Write

```
// C code:

int a, b, i, dummy, c[BUFSIZE];

if (a < b) {
   c[i++] = a;
}

;--------------
; Assembly code:

lea esi, [dummy]   ; &dummy
xor ecx, ecx       ; i = 0
...
lea    edi, [c+ecx*4]   ; &c[i]
lea    edx, [ecx+1]     ; i++
cmp    eax, ebx         ; a < b ?
cmovge edi, esi         ; ptr = (a >= b) ? &dummy : &c[i]
cmovl  ecx, edx         ; a < b ? i : i + 1
mov    [edi], eax       ; *ptr = a
```

6.4 Pairing CALL and RETURN

Optimization

Always use care when pairing CALLs and RETURNs.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When the 12-entry return-address stack gets out of synchronization, the latency of returns increases. The return-address stack becomes unsynchronized when:

- Calls and returns do not match.
- The depth of the return-address stack is exceeded because of too many levels of nested function calls.

6.5 Recursive Functions

Optimization

Use care when writing recursive functions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Returns are predicted as described in “Pairing CALL and RETURN,” so recursive functions should be written carefully. If there are only recursive function calls within the function as shown in the following example, the return address for each iteration of the recursive function is properly predicted.

Preferred

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

If there are any other calls within the recursive function (except to itself) as shown in the next example, some returns can be mispredicted. If the number of recursive function calls plus the number of nonrecursive function calls within the recursive function is greater than 12, the return stack does not predict the correct return address for some of the returns once the recursion begins to unwind.

Avoid

```
long fac(long a)
{
   if (a == 0) {
      return (1);
   } else {
      myp(a);      // Can cause returns to be mispredicted
      return (a * fac(a - 1));
   }
}

void myp(long a)
{
   printf("myp ");
   return;
}
```

Because the function `fac`, in the following example, is end-recursive, it can be converted to iterative code. A recursive function is classified as end-recursive when the function call to itself is at the end of the code. The following listing shows the rewritten code:

Preferred

```
long fac1(long a)
{
   long t = 1;
   while (a > 0) {
      myp(a);
      t *= a;
      a--;
   }
   return (t);
}
```

6.6 Nonzero Code-Segment Base Values

Optimization

In 32-bit threads, avoid using a nonzero code-segment (CS) base value. (In 64-bit mode, segmentation is disabled and the segment base value is ignored and treated as zero.)

Application

This optimization applies to:

- 32-bit software

Rationale

A nonzero CS base value causes an additional two cycles of branch-misprediction penalty when compared with a CS base value of zero:

CS base value	Minimum branch penalty (cycles)		
	Prediction sequential	Prediction taken	Misprediction
0	0	1	10
Not 0	0	1	12

6.7 Replacing Branches with Computation

Optimization

Use computation to simulate predicted execution or conditional moves.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Branches can negatively impact the performance of code. If the body of the branch is small, you can achieve higher performance by replacing the branch with computation. The computation simulates predicated execution or conditional moves. There are many SSE and SSE2 instructions that can be useful for accomplishing this. The principal instructions are as follows: ANDPS, ANDPD, ANDNPS, ANDNPD, CMPPS, CMPSS, CMPPD, CMPSD, MINPS, MINSS, MINPD, MINSD, MAXPS, MAXSS, MAXPD, MAXSD, ORPS, ORPD, PAND, PANDN, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PMAXSW, PMAXUB, PMINSW, PMINUB, POR, PXOR, XORPS, and XORPD.

For 32-bit code using 3DNow!™ instructions, try to avoid moving the MMX™ data to integer registers to perform comparisons and branches. Moving MMX data to integer registers requires either transport through memory or the use of MOVD *reg*, *mmreg* instructions, which are relatively inefficient. When using 3DNow! technology and MMX registers, the following instructions may be useful for eliminating branches: PCMPGTB, PCMPGTD, PCMPGTW, PFCMPGT, PFCMPGE, PFMIN, PFMAX, PAND, PANDN, POR, and PXOR.

Muxing Constructs

The most important construct to use in avoiding branches in SIMD code is a two-way muxing construct that is equivalent to the ternary operator (`?:`) in C and C++.

Examples

SSE Solution (Preferred)

```
; r = (x < y) ? a : b
;
; In:  XMM0 = a
;      XMM1 = b
;      XMM2 = x
;      XMM3 = y
; Out: XMM0 = r

cmpps  xmm2, xmm3, 1   ; x < y ? 0xffffffff : 0
andps  xmm0, xmm2      ; x < y ? a : 0
andnps xmm2, xmm1      ; x < y ? 0 : b
orps   xmm0, xmm2      ; x < y ? a : b
```

MMX™ Solution (Avoid)

```
; r = (x < y) ? a : b
;
; In:  MM0 = a
;      MM1 = b
;      MM2 = x
;      MM3 = y
; Out: MM0 = r

pcmpgtd mm3, mm2   ; y > x ? 0xffffffff : 0
movq    mm4, mm3   ; Duplicate mask
pandn   mm3, mm1   ; y > x ? 0 : b
pand    mm0, mm4   ; y > x ? a : 0
por     mm0, mm3   ; r = y > x ? a : b
```

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

MMX™ Solution (Preferred)

```
; r = (x < y) ? a : b
;
; In:  MM0 = a
;      MM1 = b
;      MM2 = x
;      MM3 = y
; Out: MM0 = r

pcmpgtd mm3, mm2   ; y > x ? 0xffffffff : 0
pand    mm0, mm3   ; y > x ? a: 0
pandn   mm3, mm1   ; y > x > 0 : b
por     mm0, mm3   ; r = y > x ? a : b
```

Sample Code Translated into AMD64 Code

The following examples use scalar code translated into AMD64 code. Note that it is not recommended that you 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 identically 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;
}
```

Example 1: 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;
}
```

Example 2: 3DNow!™ Code

```
; In:  MM0 = x
; Out: MM0 = z

movq     mm5, mabs   ; 0x7fffffff
pand     mm0, mm5    ; z = abs(x)
pfrcp    mm2, mm0    ; 1 / z approximation
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;
}
```

Example 3: 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;
}
```

Example 4: 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;
}
```

Example 5: 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
```

6.8 The LOOP Instruction

Optimization

Avoid using the LOOP instruction.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The LOOP instruction has a latency of at least 8 cycles.

Example

Avoid code like this, which uses the LOOP instruction:

```
label:
   ...
   loop label
```

Instead, replace the loop instruction with a DEC and a JNZ:

```
label:
   ...
   dec rcx
   jnz label
```

6.9 Far Control-Transfer Instructions

Optimization

Use far control-transfer instructions only when necessary. (Far control-transfer instructions include the far forms of JMP, CALL, and RET, as well as the INT, INTO, and IRET instructions.)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The processor's branch-prediction unit, which is used for both conditional and unconditional branches, does not predict far branches.

Chapter 7 Scheduling Optimizations

The optimizations discussed in this chapter help improve scheduling in the processor.

This chapter covers the following topics:

7.1 Instruction Scheduling by Latency

Optimization

In general, select instructions with shorter latencies that are DirectPath—not VectorPath—instructions. For a list of instruction latencies and classifications, see Appendix C, “Instruction Latencies.”

The AMD Athlon™ 64 and AMD Opteron™ processors can execute up to three AMD64 instructions per cycle, with each instruction possibly having a different latency. The AMD Athlon 64 and AMD Opteron processors have flexible scheduling, but for absolute maximum performance, schedule instructions according to their latencies and data dependencies. The goal is to reduce the overall length of dependency chains.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

7.2 Loop Unrolling

Optimization

Use loop unrolling where appropriate to increase instruction-level parallelism:

If all of these conditions are true	Then use
• The loop is in a frequently executed piece of code. • The number of loop iterations is known at compile time. • The loop body includes fewer than 10 instructions.	Complete loop unrolling
• Spare registers are available (for example, when operating in 64-bit mode, where additional registers are available). • The loop body is small, so that loop overhead is significant. • The number of loop iterations is likely greater than 10.	Partial loop unrolling

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Loop Unrolling

Loop unrolling is a technique that duplicates the body of a loop one or more times in order to increase the number of instructions relative to the branch and allow operations from different loop iterations to execute in parallel.

There are two types of loop unrolling:

- Complete loop unrolling
- Partial loop unrolling

Complete Loop Unrolling

Complete loop unrolling eliminates the loop overhead completely by replacing the loop with copies of the loop body.

Because complete loop unrolling removes the loop counter, it also reduces register pressure. However, completely unrolling very large loops can result in the inefficient use of the L1 instruction cache.

Example: Complete Loop Unrolling

In the following C code, the number of loop iterations is known at compile time and the loop body is less than 100 instructions:

```
#define ARRAY_LENGTH 3

int sum, i, a[ARRAY_LENGTH];

...
sum = 0;
for (i = 0; i < ARRAY_LENGTH; i++) {
   sum = sum + a[i];
}
```

To completely unroll an *n*-iteration loop, remove the loop control and replicate the loop body *n* times:

```
sum = 0;
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
```

Partial Loop Unrolling

Partial loop unrolling reduces the loop overhead by duplicating the loop body several times, changing the increment in the loop, and adding cleanup code to execute any leftover iterations of the loop. The number of times the loop body is duplicated is known as the *unroll factor.*

However, partial loop unrolling may increase register pressure.

Example: Partial Loop Unrolling

In the following C code, each element of one array is added to the corresponding element of another array:

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

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

Without loop unrolling, this is the equivalent assembly-language code:

```
   mov ecx, MAX_LENGTH   ; Initialize counter.
   mov eax, OFFSET a     ; Load address of array a into EAX.
   mov ebx, OFFSET b     ; Load address of array b into EBX.

add_loop:
   fld  QWORD PTR [eax]   ; Push object pointed to by EAX onto the FP stack.
   fadd QWORD PTR [ebx]   ; Add object pointed to by EBX to ST(0).
   fstp QWORD PTR [eax]   ; Copy ST(0) to object pointed to by EAX; pop ST(0).
   add  eax, 8            ; Point to next element of array a.
   add  ebx, 8            ; Point to next element of array b.
   dec  ecx               ; Decrement counter.
   jnz  add_loop          ; If elements remain, then jump.
```

The rolled loop consists of seven instructions. AMD Athlon 64 and AMD Opteron processors can decode and retire as many as three instructions per cycle, so it cannot execute faster than three iterations in seven cycles (3/7 of a floating-point add per cycle). However, the pipelined floating-point adder allows one add every cycle.

$$\frac{3\ \text{instructions}}{\text{cycle}} \times \frac{\text{iteration}}{7\ \text{instructions}} \times \frac{1\ \text{FADD}}{\text{iteration}} = \frac{3\ \text{FADDs}}{7\ \text{cycles}} = 0.429\ \text{FADDs/cycle}$$

After partial loop unrolling using an unroll factor of two, the new code creates a potential end case that must be handled outside the loop:

```
   mov ecx, MAX_LENGTH   ; Initialize counter.
   mov eax, OFFSET a     ; Load address of array a into EAX.
   mov ebx, OFFSET b     ; Load address of array b into EBX.

   shr  ecx, 1            ; Divide counter by 2 (the unroll factor).
   jnc  add_loop          ; If original counter was even, then jump.
   ; Handle the end case.
   fld  QWORD PTR [eax]   ; Push object pointed to by EAX onto the FP stack.
   fadd QWORD PTR [ebx]   ; Add object pointed to by EBX to ST(0).
   fstp QWORD PTR [eax]   ; Copy ST(0) to object pointed to by EAX; pop ST(0).
   add  eax, 8            ; Point to next element of array a.
   add  ebx, 8            ; Point to next element of array b.

add_loop:
   fld  QWORD PTR [eax]     ; Push object pointed to by EAX onto the FP stack.
   fadd QWORD PTR [ebx]     ; Add object pointed to by EBX to ST(0).
   fstp QWORD PTR [eax]     ; Copy ST(0) to object pointed to by EAX; pop ST(0).
   fld  QWORD PTR [eax+8]   ; Repeat for next element.
   fadd QWORD PTR [ebx+8]
   fstp QWORD PTR [eax+8]
   add  eax, 16             ; Point to next element of array a.
   add  ebx, 16             ; Point to next element of array b.
   dec  ecx                 ; Decrement counter.
   jnz  add_loop            ; If elements remain, then jump.
```

The unrolled loop consists of 10 instructions. Based on the decode/retire bandwidth of three instructions per cycle, this loop goes no faster than three iterations in 10 cycles (which is equivalent to 6/10 of a floating-point add per cycle because there are two additions per iteration), or 1.4 times as fast as the original loop.

$$\frac{3 \text{ instructions}}{\text{cycle}} \times \frac{\text{iteration}}{10 \text{ instructions}} \times \frac{2 \text{ FADDs}}{\text{iteration}} = \frac{6 \text{ FADDs}}{10 \text{ cycles}} = 0.600 \text{ FADDs/cycle}$$

Deriving the 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 (`low`), increases by some fixed, positive increment (`inc`) for each iteration of the loop, and may not exceed some upper bound (`high`):

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

The following code shows how to partially unroll such a loop by an unroll factor (`factor`) and how to derive the loop control for the partially unrolled version of the loop:

```
for (k = low; k <= (high - (factor - 1) * inc); k += factor * inc) {
   // Begin the series of unrolled statements.
   x[k + 0 * inc] = ...
   // Continue the series if the unrolling factor is greater than 2.
   x[k + 1 * inc] = ...
   x[k + 2 * inc] = ...
   ...
   // End the series.
   x[k + (factor - 1) * inc] = ...
}

// Handle the end cases.
for (k = k; k <= high; k += inc) {
   x[k] = ...
}
```

Related Information

For information on loop unrolling at the C-source level, see “Unrolling Small Loops” on page 13.

7.3 Inline Functions

Optimization

Use function inlining when:

- A function 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.)
- A function—once inlined—contains fewer than 25 machine instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

There are advantages and disadvantages to function inlining. On the one hand, function inlining eliminates function-call overhead and allows better register allocation and instruction scheduling at the site of the function call. The disadvantage of function inlining is decreased code reference locality, which can increase execution time due to instruction cache misses.

For functions that create fewer than 25 machine instructions once inlined, it's likely that the function-call overhead is close to, or more than, the time spent executing the function body. In these cases, function inlining is recommended.

Function-call overhead on the AMD Athlon 64 and AMD Opteron processors can be low because calls and returns are executed very quickly due to the use of prediction mechanisms. However, there is still overhead due to passing function arguments through memory, which creates store-to-load-forwarding dependencies. (In 64-bit mode, this overhead is typically avoided by passing more arguments in registers, as specified in the *AMD64 Application Binary Interface* [ABI] for the operating system.)

For longer functions, the benefits of reduced function-call overhead give diminishing returns. 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.

Additional Recommendations

In general, function inlining works best if the compiler utilizes feedback from a profiler to identify the function calls most frequently executed. If such data is not available, a reasonable approach is to concentrate on function calls inside loops. Do not consider as candidates for inlining any functions that are directly recursive. However, if they are end-recursive, the compiler should convert them to an iterative equivalent to avoid potential overflow of the processor's 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 intrinsic functions for commonly used library routines, such as `sin`, `strcmp`, or `memcpy`.

7.4 Address-Generation Interlocks

Optimization

Avoid address-generation interlocks by scheduling loads and stores whose addresses can be calculated quickly ahead of loads and stores that require the resolution of a long dependency chain in order to generate their addresses.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Address-Generation Interlocks

An *address-generation interlock* is a condition in which newer loads and stores whose addresses have already been calculated by the processor are blocked by older loads and stores whose addresses have not yet been calculated.

Rationale

The processor schedules instructions that access the data cache (loads and stores) in program order. By carefully choosing the order of loads and stores, you can avoid address-generation interlocks.

Example

Avoid code that places a load whose address takes longer to calculate before a load whose address can be determined more quickly:

```
add ebx, ecx                    ; Instruction 1
mov eax, DWORD PTR [10h]        ; Instruction 2 (fast address calc.)
mov ecx, DWORD PTR [eax+ebx]    ; Instruction 3 (slow address calc.)
mov edx, DWORD PTR [24h]        ; This load is stalled from accessing the
                                ;  data cache due to the long latency
                                ;  caused by generating the address for
                                ;  instruction 3.
```

Where possible, reorder instructions so that loads with simpler address calculations come before those with more complex address calculations:

```
add ebx, ecx                    ; Instruction 1
mov eax, DWORD PTR [10h]        ; Instruction 2
mov edx, DWORD PTR [24h]        ; Place load above instruction 3 to avoid
                                ;  address-generation interlock stall.
mov ecx, DWORD PTR [eax+ebx]    ; Instruction 3
```

7.5 MOVZX and MOVSX

Optimization

Use the MOVZX and MOVSX instructions to zero-extend or sign-extend, respectively, an operand to a larger size.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Typical code for zero extension that replaces MOVZX uses more decode and execution resources than MOVZX. It also has higher latency due to the superset dependency between the XOR and the MOV, which requires a merge operation.

Example

When zero-extending an operand (in this case, a byte), avoid code such as the following:

```
xor rax, rax
mov al, mem
```

Instead, use the MOVZX instruction:

```
movzx rax, BYTE PTR mem
```

7.6 Pointer Arithmetic in Loops

Optimization

Minimize pointer arithmetic in loops, especially if the loop bodies are small. Take advantage of scaled-index addressing modes to utilize the loop counter as an index into memory arrays.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In small loops, pointer arithmetic causes significant overhead. Using scaled-index addressing modes has no negative impact on execution speed, but the reduced number of instructions preserves decode bandwidth.

Example

Consider the following C code, which adds the elements of two arrays and stores them in a third array:

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

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

Avoid an assembly-language equivalent like this, which uses base and displacement components (for example, [esi+a]) to compute array-element addresses, requiring additional pointer arithmetic to increment the offsets into the forward-traversed arrays:

```
   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 from a.
   mov edx, [edi+b]   ; Get element from 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 is 0.
```

Instead, traverse the arrays in a downward direction (from higher to lower addresses), in order to take advantage of scaled-index addressing (for example, [ecx*4+a]), which minimizes pointer arithmetic within the loop:

```
   mov ecx, MAXSIZE - 1   ; Initialize index.

add_loop:
   mov eax, [ecx*4+a]   ; Get element from a.
   mov edx, [ecx*4+b]   ; Get element from 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 is negative.
```

A change in the direction of traversal is possible only if each loop iteration is completely independent of the others. If you can't change the direction of traversal for a given array, it's still possible to minimize pointer arithmetic by using as a base address a displacement that points to the byte past the end of the array, and using an index that starts with a negative value and reaches zero when the loop expires:

```
   mov ecx, (-MAXSIZE)   ; Initialize index.

add_loop:
   mov eax, [ecx*4+a+MAXSIZE*4]   ; Get element from a.
   mov edx, [ecx*4+b+MAXSIZE*4]   ; Get element from b.
   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 is 0.
```

If the base addresses of the arrays are held in registers (for example, when the base addresses are passed as the 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.

7.7 Pushing Memory Data Directly onto the Stack

Optimization

Push memory data directly onto the stack instead of loading it into a register first.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Pushing memory data directly onto the stack reduces register pressure and eliminates data dependencies.

Example

Avoid code that first loads the memory data into a register and then pushes it onto the stack:

```
mov  rax, mem
push rax
```

Instead, push the memory data directly onto the stack:

```
push mem
```

Chapter 8 Integer Optimizations

The optimizations in this chapter help improve integer performance.

This chapter covers the following topics:

8.1 Replacing Division with Multiplication

Optimization

Replace integer division by constants with multiplication by the reciprocal.

Rationale

Because the AMD Athlon™ 64 and AMD Opteron™ processors have very fast integer multiplication (3–8 cycles signed, 3–8 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. Either follow the examples in this chapter that illustrate the use of integer division by constants or create the executables using the code in "Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants" on page 186.

Multiplication by Reciprocal (Division) Utility

The code for the utilities is shown in "Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants" on page 186. The utilities provided in this document are for reference only and are not supported by AMD.

Signed Division Utility

The `sdiv.exe` utility finds the fastest code for *signed* division by a constant. The utility displays the code after the user enters a signed constant divisor. To redirect the code to a file, type the following command:

```
sdiv > example.out
```

Unsigned Division Utility

The `udiv.exe` utility finds the fastest code for *unsigned* division by a constant. The utility displays the code after the user enters an unsigned constant divisor. To redirect the code to a file, type the following command:

```
udiv > example.out
```

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.

```
; a = algorithm
; m = multiplier
; s = shift factor

; a == 0
mov eax, m
mul dividend
shr edx, s   ; EDX = quotient

; a == 1
mov eax, m
mul dividend
add eax, m
adc edx, 0
shr edx, s   ; EDX = quotient
```

Code for determining the algorithm (*a*), multiplier (*m*), and shift factor (*s*) from the divisor (*d*) is found in the section “Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants” on page 186.

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

For divisors $2^{31} <= d < 2^{32}$, the possible quotient values are either 0 or 1. For this reason, it is easy to establish the quotient by simple comparison of the dividend and divisor.When the dividend needs to be preserved, consider using code like the following:

```
; In:  EAX = dividend
; Out: EDX = quotient

xor edx, edx   ; 0
cmp eax, d     ; CF = (dividend < divisor) ? 1 : 0
sbb edx, -1    ; quotient = 0 + 1 - CF = (dividend < divisor) ? 0 : 1
```

When the dividend does not need to be preserved, the division can be accomplished without the use of an additional register, thus reducing register pressure, as shown here:

```
; In:  EAX = dividend
; Out: EDX = quotient

cmp edx, d    ; CF = (dividend < divisor) ? 1 : 0
mov eax, 0    ; 0
sbb eax, -1   ; quotient = 0 + 1 - CF = (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 4000_0005h:

```
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` instruction to the code. These changes make use of the fact that $n/–d = –(n/d)$.

```
; a = algorithm
; m = multiplier
; s = shift count

; a == 0
mov  eax, m
imul dividend
mov  eax, dividend
shr  eax, 31
sar  edx, s
add  edx, eax   ; Quotient in EDX

; a == 1
mov  eax, m
imul dividend
mov  eax, dividend
add  edx, eax
shr  eax, 31
sar  edx, s
add  edx, eax   ; Quotient in EDX
```

Code for determining the algorithm (*a*), multiplier (*m*), and shift factor (*s*) is shown in “Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants” on page 186.

Signed Division by 2

```
; In:  EAX = dividend
; Out: EAX = quotient

cmp eax, 80000000h   ; CF = 1 if dividend >= 0.
sbb eax, -1          ; Increment dividend if it is < 0.
sar eax, 1           ; Perform 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, 80000000h   ; CF = 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 Division by 2 or –2

```
; In:  EAX = dividend
; Out: EAX = remainder

cdq            ; Sign extend into EDX.
and eax, 1     ; Compute remainder.
xor eax, edx   ; Negate remainder if
sub eax, edx   ;  dividend was < 0.
```

Remainder of Signed Division by 2^n or $-(2^n)$

```
; In:  EAX = dividend
; Out: EAX = remainder

cdq                  ; Sign extend into EDX.
and edx, (2^n - 1)   ; Mask correction (abs(divisor) - 1)
add eax, edx         ; Apply pre-correction.
and eax, (2^n - 1)   ; Mask out remainder (abs(divisor) - 1)
sub eax, edx         ; Apply pre-correction if necessary.
```

8.2 Alternative Code for Multiplying by a Constant

Optimization

Devise instruction sequences with lower latency to accomplish multiplication by certain constant multipliers.

Rationale

A 32-bit integer multiplied by a constant has a latency of 3 cycles; a 64-bit integer multiplied by a constant has a latency of 4 cycles. For certain constant multipliers, instruction sequences can be devised that accomplish the multiplication with lower latency. Because the AMD Athlon 64 and AMD Opteron processors contain only one integer multiplier but three integer execution units, the replacement code can provide better throughput as well.

Most replacement sequences require the use of an additional temporary register, thus increasing register pressure. If register pressure in a piece of code that performs integer multiplication with a constant is already high, it could be better for the overall performance of that code to use the IMUL instruction instead of the replacement code. Similarly, replacement sequences with low latency but containing many instructions may negatively influence decode bandwidth as compared to the IMUL instruction. In general, replacement sequences containing more than four instructions are not recommended.

The following code samples are designed for the original source to receive the final result. Other sequences are possible if the result is in a different register. Sequences that do not require a temporary register are favored over ones requiring a temporary register, even if the latency is higher. Arithmetic-logic-unit operations are preferred over shifts to keep code size small. Similarly, both arithmetic-logic-unit operations and shifts are favored over the LEA instruction.

There are improvements in the AMD Athlon 64 and AMD Opteron processors' multiplier over that of previous x86 processors. For this reason, when doing 32-bit multiplication, only use the alternative sequence if the alternative sequence has a latency that is less than or equal to 2 cycles. For 64-bit multiplication, only use the alternative sequence if the alternative sequence has a latency that is less than or equal to 3 cycles.

Examples

```
by 2:   add reg1, reg1            ; 1 cycle

by 3:   lea reg1, [reg1+reg1*2]   ; 2 cycles

by 4:   shl reg1, 2               ; 1 cycle

by 5:   lea reg1, [reg1+reg1*4]   ; 2 cycles
```

```
by 6:   lea reg1, [reg1+reg1*2]   ; 3 cycles
        add reg1, reg1

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+reg1*8]   ; 2 cycles

by 10:  lea reg1, [reg1+reg1*4]   ; 3 cycles
        add reg1, reg1

by 11:  lea reg2, [reg1+reg1*8]   ; 3 cycles
        add reg1, reg1
        add reg1, reg2

by 12:  lea reg1, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 2

by 13:  lea reg2, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 4
        sub reg1, reg2

by 14:  lea reg2, [reg1+reg1]     ; 3 cycles
        shl reg1, 4
        sub reg1, reg2

by 15:  mov reg2, reg1            ; 3 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:  lea reg1, [reg1+reg1*8]   ; 3 cycles
        add reg1, reg1

by 19:  lea reg2, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 4
        add reg1, reg2

by 20:  lea reg1, [reg1+reg1*4]   ; 3 cycles
        shl reg1, 2

by 21:  lea reg2, [reg1+reg1*4]   ; 3 cycles
        shl reg1, 4
```

```
        add reg1, reg2

by 22:  imul reg1, 22             ; Use the IMUL instruction.

by 23:  lea reg2, [reg1+reg1*8]   ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 24:  lea reg1, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 3

by 25:  lea reg2, [reg1+reg1*8]   ; 3 cycles
        shl reg1, 4
        add reg1, reg2

by 26:  imul reg1, 26             ; Use the IMUL instruction.

by 27:  lea reg2, [reg1+reg1*4]   ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 28:  lea reg2, [REG1*4]        ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 29:  lea reg2, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 30:  lea reg2, [reg1+reg1]     ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 31:  mov reg2, reg1            ; 2 cycles
        shl reg1, 5
        sub reg1, reg2

by 32:  shl reg1, 5               ; 1 cycle
```

8.3 Repeated String Instructions

Optimization

Avoid using the REP prefix when performing string operations, especially when copying blocks of memory.

Rational

In general, using the REP prefix to repeatedly perform string instructions is less optimal than other methods, especially when copying blocks of memory. For a discussion of alternate memory-copy methods, see “Appropriate Memory Copying Routines” on page 112.

Latency of Repeated String Instructions

Table 7 shows the latency of repeated string instructions on the AMD Athlon 64 and AMD Opteron processors.

Table 7 lists the latencies with the direction flag (DF) = 0 (increment) and DF = 1 (decrement). In addition, these latencies are assumed for aligned memory operands. Note that for MOVS and STOS, when DF = 1, 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.

Table 7. Latency of Repeated String Instructions

	Number of Cycles		
Instruction	**When ECX = 0**	**When ECX = c[1], DF = 0**	**When ECX = c[1], DF = 1**
`rep movs`	11	15 + (1 * 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:
1. *c > 0*

Guidelines for Repeated String Instructions

To help achieve good performance, the following sections contain 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 STOSB`.

In 64-bit mode, a quadword data size is available and offers better performance (for example, `REP MOVSQ` and `REP STOSQ`).

Ensure DF = 0 (Increment)

Always make sure that DF is 0 (increment) after execution of CLD for `rep movs` and `rep stos`. DF = 1 (decrement) is only needed for certain cases of overlapping `rep movs` (for example, source and destination overlap).

While string instructions with DF = 1 (decrement) are slower, only the overhead part of the cycle equation is larger and not the throughput part. See Table 7 on page 167 for additional latency numbers.

Align Source and Destination with Operand Size

For `rep movs`, make sure that both the 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

If the repeat count is constant and low (less than eight), expand REP string instructions into equivalent sequences of simple AMD64 instructions. 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`.

8.4 Using XOR to Clear Integer Registers

Optimization

To clear an integer register to all zeros, use the XOR instruction to exclusive OR the register with itself, as shown below.

Rationale

AMD Athlon 64 and AMD Opteron processors are able to avoid the false read dependency on the XOR instruction.

Examples

Acceptable

```
mov reg, 0
```

Preferred

```
xor reg, reg
```

8.5 Efficient 64-Bit Integer Arithmetic in 32-Bit Mode

Optimization

The following section contains a collection of code snippets and subroutines showing the efficient implementation of 64-bit arithmetic in 32-bit mode. Note that these are 32-bit recommendations, in 64-bit mode it is important to use 64-bit integer instructions for best performance.

Addition, subtraction, negation, and shifting are best handled by inline code. Multiplication, division, and the computation of remainders are less common operations and are usually implemented as subroutines. If these subroutines are used often, the programmer should consider inlining them. Except for division and remainder calculations, the following code 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.

64-Bit Addition

```
; Add ECX:EBX to EDX:EAX, and place sum in EDX:EAX.
add eax, ebx
adc edx, ecx
```

64-Bit Subtraction

```
; Subtract ECX:EBX from EDX:EAX and place difference in EDX:EAX.
sub eax, ebx
sbb edx, ecx
```

64-Bit Negation

```
; Negate EDX:EAX.
not edx
neg eax
sbb edx, -1   ; Fix: Increment high word if low word was 0.
```

64-Bit Left Shift

```
; Shift 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:
```

64-Bit 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:
```

64-Bit Multiplication

```
; _llmul computes the low-order half of the product of its
;  arguments, two 64-bit integers.
;
; In:       [ESP+8]:[ESP+4] = multiplicand
;           [ESP+16]:[ESP+12] = multiplier
; Out:      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 + p2_lo + p3_lo = result in EDX:EAX
   ret                       ; Done, return to caller.

_llmul ENDP
```

64-Bit Unsigned Division

```
; _ulldiv divides two unsigned 64-bit integers and returns the quotient.
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
; Out:      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 > 2^32 – 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            ;  2^32 (that is, it 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 high word (low only)
   mul  dword ptr [esp+20] ; quotient * divisor low 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 high 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
```

64-Bit Signed Division

```
; _lldiv divides two signed 64-bit numbers and delivers the quotient
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
; Out:      EDX:EAX = quotient of division
; Destroys: EAX, ECX,E DX, EFlags

_lldiv PROC
   push ebx    ; Save EBX as per calling convention.
   push esi    ; Save ESI as per calling convention.
   push edi    ; Save EDI as per calling convention.
   mov  ecx, [esp+28]   ; divisor_hi
   mov  ebx, [esp+24]   ; divisor_lo
   mov  edx, [esp+20]   ; dividend_hi
   mov  eax, [esp+16]   ; dividend_lo
   mov  esi, ecx        ; divisor_hi
   xor  esi, edx        ; divisor_hi ^ dividend_hi
   sar  esi, 31         ; (quotient < 0) ? -1 : 0
   mov  edi, edx        ; dividend_hi
   sar  edi, 31         ; (dividend < 0) ? -1 : 0
   xor  eax, edi        ; If (dividend < 0),
   xor  edx, edi        ;  compute 1's complement of dividend.
   sub  eax, edi        ; If (dividend < 0),
   sbb  edx, edi        ;  compute 2's complement of dividend.
   mov  edi, ecx        ; divisor_hi
   sar  edi, 31         ; (divisor < 0) ? -1 : 0
   xor  ebx, edi        ; If (divisor < 0),
   xor  ecx, edi        ;  compute 1's complement of divisor.
   sub  ebx, edi        ; If (divisor < 0),
   sbb  ecx, edi        ;  compute 2's complement of divisor.
   jnz  big_divisor     ; divisor > 2^32 - 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)
   xor  eax, esi        ; If (quotient < 0),
   xor  edx, esi        ;  compute 1's complement of result.
   sub  eax, esi        ; If (quotient < 0),
   sbb  edx, esi        ;  compute 2's complement of result.
   pop  edi             ; Restore EDI as per calling convention.
   pop  esi             ; Restore ESI as per calling convention.
   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)
   jmp  make_sign   ; Make quotient signed.

big_divisor:
   sub  esp, 12             ; Create three local variables.
   mov  [esp], eax          ; dividend_lo
   mov  [esp+4], ebx        ; divisor_lo
   mov  [esp+8], edx        ; dividend_hi
   mov  edi, ecx            ; Save divisor_hi.
   shr  edx, 1              ; Shift both
   rcr  eax, 1              ;  divisor and
   ror  edi, 1              ;  and dividend
   rcr  ebx, 1              ;  right by 1 bit.
   bsr  ecx, ecx            ; ECX = number of remaining shifts
   shrd ebx, edi, cl        ; Scale down divisor and
   shrd eax, edx, cl        ;  dividend such that divisor is
   shr  edx, cl             ;  less than 2^32 (that is, fits in EBX).
   rol  edi, 1              ; Restore original divisor_hi.
   div  ebx                 ; Compute quotient.
   mov  ebx, [esp]          ; dividend_lo
   mov  ecx, eax            ; Save quotient.
   imul edi, eax            ; quotient * divisor high word (low only)
   mul  DWORD PTR [esp+4]   ; quotient * divisor low word
   add  edx, edi            ; EDX:EAX = quotient * divisor
   sub  ebx, eax            ; dividend_lo - (quot.*divisor)_lo
   mov  eax, ecx            ; Get quotient.
   mov  ecx, [esp+8]        ; dividend_hi
   sbb  ecx, edx            ; Subtract (divisor * quot.) from dividend
   sbb  eax, 0              ; Adjust quotient if remainder is negative.
   xor  edx, edx            ; Clear high word of quotient.
   add  esp, 12             ; Remove local variables.

make_sign:
   xor eax, esi   ; If (quotient < 0),
   xor edx, esi   ;  compute 1's complement of result.
   sub eax, esi   ; If (quotient < 0),
   sbb edx, esi   ;  compute 2's complement of result.
   pop edi        ; Restore EDI as per calling convention.
   pop esi        ; Restore ESI as per calling convention.
   pop ebx        ; Restore EBX as per calling convention.
   ret            ; Done, return to caller.
_lldiv ENDP
```

64-Bit Unsigned Remainder Computation

```
; _ullrem divides two unsigned 64-bit integers and returns the remainder.
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
;
; Out:      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 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            ;  (that is, it fits in EBX).
   rol  edi, 1             ; Restore original divisor (EDI:ESI).
   div  ebx                ; Compute quotient.
   mov  ebx, [esp+12]      ; dividend low word
   mov  ecx, eax           ; Save quotient.
   imul edi, eax           ; quotient * divisor high word (low only)
   mul  DWORD PTR [esp+20] ; quotient * divisor low 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) ? 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
```

64-Bit Signed Remainder Computation

```
; _llrem divides two signed 64-bit numbers and returns the remainder.
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
;
; Out:      EDX:EAX = remainder of division
;
; Destroys: EAX, ECX, EDX, EFlags

   push ebx              ; Save EBX as per calling convention.
   push esi              ; Save ESI as per calling convention.
   push edi              ; Save EDI as per calling convention.
   mov  ecx, [esp+28]    ; divisor-hi
   mov  ebx, [esp+24]    ; divisor-lo
   mov  edx, [esp+20]    ; dividend-hi
   mov  eax, [esp+16]    ; dividend-lo
   mov  esi, edx         ; sign(remainder) == sign(dividend)
   sar  esi, 31          ; (remainder < 0) ? -1 : 0
   mov  edi, edx         ; dividend-hi
   sar  edi, 31          ; (dividend < 0) ? -1 : 0
   xor  eax, edi         ; If (dividend < 0),
   xor  edx, edi         ;  compute 1's complement of dividend.
   sub  eax, edi         ; If (dividend < 0),
   sbb  edx, edi         ;  compute 2's complement of dividend.
   mov  edi, ecx         ; divisor-hi
   sar  edi, 31          ; (divisor < 0) ? -1 : 0
   xor  ebx, edi         ; If (divisor < 0),
   xor  ecx, edi         ;  compute 1's complement of divisor.
   sub  ebx, edi         ; If (divisor < 0),
   sbb  ecx, edi         ;  compute 2's complement of divisor.
   jnz  sr_big_divisor   ; divisor > 2^32 - 1
   cmp  edx, ebx         ; Only one division needed (ECX = 0)?
   jae  sr_two_divs      ; No, need two divisions.
   div  ebx              ; EAX = quotient_lo
   mov  eax, edx         ; EAX = remainder_lo
   mov  edx, ecx         ; EDX = remainder_lo = 0
   xor  eax, esi         ; If (remainder < 0),
   xor  edx, esi         ;  compute 1's complement of result.
```

```
   sub  eax, esi          ; If (remainder < 0),
   sbb  edx, esi          ;  compute 2's complement of result.
   pop  edi               ; Restore EDI as per calling convention.
   pop  esi               ; Restore ESI as per calling convention.
   pop  ebx               ; Restore EBX as per calling convention.
   ret                    ; Done, return to caller.

sr_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       ; remainder_lo
   xor  edx, edx       ; remainder_hi = 0
   jmp  sr_makesign    ; Make remainder signed.

sr_big_divisor:
   sub  esp, 16             ; Create three local variables.
   mov  [esp], eax          ; dividend_lo
   mov  [esp+4], ebx        ; divisor_lo
   mov  [esp+8], edx        ; dividend_hi
   mov  [esp+12], ecx       ; divisor_hi
   mov  edi, ecx            ; Save divisor_hi.
   shr  edx, 1              ; Shift both
   rcr  eax, 1              ;  divisor and
   ror  edi, 1              ;  and dividend
   rcr  ebx, 1              ;  right by 1 bit.
   bsr  ecx, ecx            ; ECX = number of remaining shifts
   shrd ebx, edi, cl        ; Scale down divisor and
   shrd eax, edx, cl        ;  dividend such that divisor is
   shr  edx, cl             ;  less than 2^32 (that is, fits in EBX).
   rol  edi, 1              ; Restore original divisor_hi.
   div  ebx                 ; Compute quotient.
   mov  ebx, [esp]          ; dividend_lo
   mov  ecx, eax            ; Save quotient.
   imul edi, eax            ; quotient * divisor high word (low only)
   mul  DWORD PTR [esp+4]   ; quotient * divisor low word
   add  edx, edi            ; EDX:EAX = quotient * divisor
   sub  ebx, eax            ; dividend_lo - (quot.*divisor)_lo
   mov  ecx, [esp+8]        ; dividend_hi
   sbb  ecx, edx            ; Subtract divisor * quot. from dividend.
   sbb  eax, eax            ; remainder < 0 ? 0xffffffff : 0
   mov  edx, [esp+12]       ; divisor_hi
   and  edx, eax            ; remainder < 0 ? divisor_hi : 0
   and  eax, [esp+4]        ; remainder < 0 ? divisor_lo : 0
   add  eax, ebx            ; remainder_lo
   add  edx, ecx            ; remainder_hi
   add  esp, 16             ; Remove local variables.
```

```
sr_makesign:
   xor eax, esi   ; If (remainder < 0),
   xor edx, esi   ;  compute 1's complement of result.
   sub eax, esi   ; If (remainder < 0),
  sbb edx, esi   ;  compute 2's complement of result.
   pop edi        ; Restore EDI as per calling convention.
   pop esi        ; Restore ESI as per calling convention.
   pop ebx        ; Restore EBX as per calling convention.
   ret            ; Done, return to caller.
```

8.6 Efficient Implementation of Population-Count Function in 32-Bit Mode

Population count is an operation that determines the number of set bits in a bit string. For example, this can be used to determine the cardinality of a set. The example code in this section shows how to efficiently implement a population count operation for 32-bit operands. The example is written for the inline assembler of Microsoft® Visual C.

Function `popcount` implements a branchless computation of the population count. It is based on a O(log(n)) algorithm that successively groups the bits into groups of 2, 4, 8, 16, and 32, while maintaining a count of the set bits in each group. The algorithm consists of the following steps:

1. Partition the integer into groups of two bits. Compute the population count for each 2-bit group and store the result in the 2-bit group. This calls for the following transformation to be performed for each 2-bit group:

   ```
   00b -> 00b
   01b -> 01b
   10b -> 01b
   11b -> 10b
   ```

   If the original value of a 2-bit group is *v*, then the new value will be *v* – (*v* >> 1). In order to handle all 2-bit groups simultaneously, it is necessary to mask appropriately to prevent spilling from one bit group to the next lower bit group. Thus:

   ```
   w = v - ((v >> 1) & 0x55555555)
   ```

2. Add the population count of adjacent 2-bit group and store the sum to the 4-bit group resulting from merging these adjacent 2-bit groups. To do this simultaneously to all groups, mask out the odd numbered groups, mask out the even numbered groups, and then add the odd numbered groups to the even numbered groups:

   ```
   x = (w & 0x33333333) + ((w >> 2) & 0x33333333)
   ```

   Each 4-bit field now has one of the following values: 0000b, 0001b, 0010b, 0011b, or 0100b.

3. For the first time, the value in each *k*-bit field is small enough that adding two *k*-bit fields results in a value that still fits in the *k*-bit field. Thus the following computation is performed:

   ```
   y = (x + (x >> 4)) & 0x0F0F0F0F
   ```

   The result is four 8-bit fields whose lower half has the desired sum and whose upper half contains "junk" that has to be masked out. A symbolic form is as follows:

   ```
   x      = 0aaa0bbb0ccc0ddd0eee0fff0ggg0hhh
   x >> 4 = 00000aaa0bbb0ccc0ddd0eee0fff0ggg
   sum    = 0aaaWWWWiiiiXXXXjjjjYYYYkkkkZZZZ
   ```

   The WWWW, XXXX, YYYY, and ZZZZ values are the interesting sums with each at most 1000b, or 8 decimal.

4. The four 4-bit sums can now be rapidly accumulated by multiplying with a so-called *magic* multiplier. This can be derived from looking at the following chart of partial products:

```
0p0q0r0s * 01010101 =

        :0p0q0r0s
      0p:0q0r0s
    0p0q:0r0s
  0p0q0r:0s
000pxxww:vvuutt0s
```

   Here p, q, r, and s are the 4-bit sums from the previous step, and vv is the final interesting result. The final result is as follows:

```
z = (y * 0x01010101) >> 24
```

Integer Version

```
unsigned int popcount(unsigned int v)

{
   unsigned int retVal;
   __asm {
      mov  eax, [v]          ; v
      mov  edx, eax          ; v
      shr  eax, 1            ; v >> 1
      and  eax, 055555555h   ; (v >> 1) & 0x55555555
      sub  edx, eax          ; w = v - ((v >> 1) & 0x55555555)
      mov  eax, edx          ; w
      shr  edx, 2            ; w >> 2
      and  eax, 033333333h   ; w & 0x33333333
      and  edx, 033333333h   ; (w >> 2)  & 0x33333333
      add  eax, edx          ; x = (w & 0x33333333) + ((w >> 2) &
                             ;  0x33333333)
      mov  edx, eax          ; x
      shr  eax, 4            ; x >> 4
      add  eax, edx          ; x + (x >> 4)
      and  eax, 00F0F0F0Fh   ; y = (x + (x >> 4) & 0x0F0F0F0F)
      imul eax, 001010101h   ; y * 0x01010101
      shr  eax, 24           ; population count = (y *
                             ;  0x01010101) >> 24
      mov  retVal, eax       ; Store result.
   }
   return(retVal);
}
```

8.7 Efficient Binary-to-ASCII Decimal Conversion

Fast binary-to-ASCII decimal conversion can be important to the performance of software working with text oriented protocols like HTML, such as web servers. The following examples show two optimized functions for fast conversion of unsigned integers-to-ASCII decimal strings on AMD Athlon 64 and AMD Opteron processors. The code is written for the Microsoft Visual C compiler.

The function `uint_to_ascii_lz` converts like `sprintf(sptr, "%010u", x)`. That is, leading zeros are retained, whereas `uint_to_ascii_nlz` converts like `sprintf(sptr, "%u", x)`; that is, leading zeros are suppressed.

This code can easily be extended to convert signed integers by isolating the sign information and computing the absolute value as shown in Listing on page 130 before starting the conversion process. For restricted argument ranges, construct more efficient conversion routines using the same algorithm as used for the general case presented here.

The algorithm first splits the input argument into suitably sized blocks by dividing the input by an appropriate power of ten and working separately on the quotient and remainder of that division. The DIV instruction is avoided as described in “Replacing Division with Multiplication” on page 160. Each block is then converted into a fixed-point format that consists of one (decimal) integer digit and a binary fraction. This allows the generation of additional decimal digits by repeated multiplication of the fraction by 10. For efficiency reasons the algorithm implements this multiplication by multiplying by five and moving the binary point to the right by one bit for each step of the algorithm. To avoid loop overhead and branch mispredictions, the digit generation loop is completely unrolled. In order to maximize parallelism, the code in `uint_to_ascii_lz` splits the input into two equally sized blocks each of which yields five decimal digits for the result.

Binary-to-ASCII Decimal Conversion Retaining Leading Zeros

```
__declspec(naked) void __stdcall uint_to_ascii_lz(char *sptr, unsigned int x)
{
   __asm {
      push edi                ; Save as per calling conventions.
      push esi                ; Save as per calling conventions.
      push ebx                ; Save as per calling conventions.
      mov  eax, [esp+20]      ; x
      mov  edi, [esp+16]      ; sptr
      mov  esi, eax           ; x
      mov  edx, 0xA7C5AC47    ; Divide x by
      mul  edx                ;  10,000 using
      add  eax, 0xA7C5AC47    ;  multiplication
      adc  edx, 0             ;  with reciprocal.
      shr  edx, 16            ; y1 = x / 1e5
      mov  ecx, edx           ; y1
      imul edx, 100000        ; (x / 1e5) * 1e5
      sub  esi, edx           ; y2 = x % 1e5
      mov  eax, 0xD1B71759    ; 2^15 / 1e4 * 2^30
      mul  ecx                ; Divide y1 by 1e4,
```

```
shr  eax, 30            ;  converting it into
lea  ebx, [eax+edx*4+1] ;  17.15 fixed-point format
mov  ecx, ebx           ;  such that 1.0 = 2^15.
mov  eax, 0xD1B71759    ; 2^15 / 1e4 * 2^30
mul  esi                ; Divide y2 by 1e4,
shr  eax, 30            ;  converting it into
lea  esi, [eax+edx*4+1] ;  17.15 fixed-point format
mov  edx, esi           ;  such that 1.0 = 2^15.
shr  ecx, 15            ; 1st digit
and  ebx, 0x00007fff    ; Fraction part
OR   ecx, '0'           ; Convert 1st digit to ASCII.
mov  [edi+0], cl        ; Store 1st digit in memory.
lea  ecx, [ebx+ebx*4]   ; 5 * fraction, new digit ECX[31-14]
lea  ebx, [ebx+ebx*4]   ; 5 * fraction, new fraction EBX[13-0]
shr  edx, 15            ; 6th digit
and  esi, 0x00007fff    ; Fraction part
or   edx, '0'           ; Convert 6th digit to ASCII.
mov  [edi+5], dl        ; Store 6th digit in memory.
lea  edx, [esi+esi*4]   ; 5 * fraction, new digit EDX[31-14]
lea  esi, [esi+esi*4]   ; 5 * fraction, new fraction ESI[13-0]
shr  ecx, 14            ; 2nd digit
and  ebx, 0x00003fff    ; Fraction part
or   ecx, '0'           ; Convert 2nd digit to ASCII.
mov  [edi+1], cl        ; Store 2nd digit in memory.
lea  ecx, [ebx+ebx*4]   ; 5 *f raction, new digit ECX[31-13]
lea  ebx, [ebx+ebx*4]   ; 5 * fraction, new fraction EBX[12-0]
shr  edx, 14            ; 7th digit
and  esi, 0x00003fff    ; Fraction part
or   edx, '0'           ; Convert 7th digit to ASCII.
mov  [edi+6], dl        ; Store 7th digit in memory.
lea  edx, [esi+esi*4]   ; 5 * fraction, new digit EDX[31-13]
lea  esi, [esi+esi*4]   ; 5 * fraction, new fraction ESI[12-0]
shr  ecx, 13            ; 3rd digit
and  ebx, 0x00001fff    ; Fraction part
or   ecx, '0'           ; Convert 3rd digit to ASCII.
mov  [edi+2], cl        ; Store 3rd digit in memory.
lea  ecx, [ebx+ebx*4]   ; 5 * fraction, new digit ECX[31-12]
lea  ebx, [ebx+ebx*4]   ; 5 * fraction, new fraction EBX[11-0]
shr  edx, 13            ; 8th digit
and  esi, 0x00001fff    ; Fraction part
or   edx, '0'           ; Convert 8th digit to ASCII.
mov  [edi+7], dl        ; Store 8th digit in memory.
lea  edx, [esi+esi*4]   ; 5 * fraction, new digit EDX[31-12]
lea  esi, [esi+esi*4]   ; 5 * fraction, new fraction ESI[11-0]
shr  ecx, 12            ; 4th digit
and  ebx, 0x00000fff    ; Fraction part
or   ecx, '0'           ; Convert 4th digit to ASCII.
mov  [edi+3], cl        ; Store 4th digit in memory.
lea  ecx, [ebx+ebx*4]   ; 5 * fraction, new digit ECX[31-11]
shr  edx, 12            ; 9th digit
and  esi, 0x00000fff    ; Fraction part
```

```
      or   edx, '0'           ; Convert 9th digit to ASCII.
      mov  [edi+8], dl        ; Store 9th digit in memory.
      lea  edx, [esi+esi*4]   ; 5 * fraction, new digit EDX[31-11]
      shr  ecx, 11            ; 5th digit
      or   ecx, '0'           ; Convert 5th digit to ASCII.
      mov  [edi+4], cl        ; Store 5th digit in memory.
      shr  edx, 11            ; 10th digit
      or   edx, '0'           ; Convert 10th digit to ASCII.
      mov  [edi+9], dx        ; Store 10th digit and end marker in memory.
      pop  ebx                ; Restore register as per calling convention.
      pop  esi                ; Restore register as per calling convention.
      pop  edi                ; Restore register as per calling convention.
      ret  8                  ; Pop two DWORD arguments and return.
   }
}
```

Binary-to-ASCII Decimal Conversion Suppressing Leading Zeros

```
__declspec(naked) void __stdcall uint_to_ascii_nlz(char *sptr, unsigned int x)
{
   __asm {
      push edi                ; Save as per calling conventions.
      push ebx                ; Save as per calling conventions.
      mov  edi, [esp+12]      ; sptr
      mov  eax, [esp+16]      ; x
      mov  ecx, eax           ; Save original argument.
      mov  edx, 89705F41h     ; 1e-9 * 2^61 rounded
      mul  edx                ; Divide by 1e9 by multiplying with reciprocal.
      add  eax, eax           ; Round division result.
      adc  edx, 0             ; EDX[31-29] = argument / 1e9
      shr  edx, 29            ; Leading decimal digit, 0...4
      mov  eax, edx           ; Leading digit
      mov  ebx, edx           ; Initialize digit accumulator with
                              ;  leading digit.
      imul eax, 1000000000    ; Leading digit * 1e9
      sub  ecx, eax           ; Subtract (leading digit * 1e9) from argument.
      or   dl, '0'            ; Convert leading digit to ASCII.
      mov  [edi], dl          ; Store leading digit.
      cmp  ebx, 1             ; Any nonzero digit yet?
      sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
      mov  eax, ecx           ; Get reduced argument < 1e9.
      mov  edx, 0abcc7712h    ; 2^28 / 1e8 * 2^30 rounded up
      mul  edx                ; Divide reduced
      shr  eax, 30            ;  argument < 1e9 by 1e8,
      lea  edx, [eax+4*edx+1] ;  converting it into 4.28 fixed-point
      mov  eax, edx           ;  format such that 1.0 = 2^28.
      shr  eax, 28            ; Next digit
      and  edx, 0fffffffh     ; Fraction part
      or   ebx, eax           ; Accumulate next digit.
      or   eax, '0'           ; Convert digit to ASCII.
      mov  [edi], al          ; Store digit in memory.
      lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-27]
```

```
lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[26-0]
cmp  ebx, 1             ; Any nonzero digit yet?
sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
shr  eax, 27            ; Next digit
and  edx, 07ffffffh     ; Fraction part
or   ebx, eax           ; Accumulate next digit.
or   eax, '0'           ; Convert digit to ASCII.
mov  [edi], al          ; Store digit in memory.
lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-26]
lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[25-0]
cmp  ebx, 1             ; Any nonzero digit yet?
sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
shr  eax, 26            ; Next digit
and  edx, 03ffffffh     ; Fraction part
or   ebx, eax           ; Accumulate next digit.
or   eax, '0'           ; Convert digit to ASCII.
mov  [edi], al          ; Store digit in memory.
lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-25]
lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[24-0]
cmp  ebx, 1             ; Any nonzero digit yet?
sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
shr  eax, 25            ; Next digit
and  edx, 01ffffffh     ; Fraction part
or   ebx, eax           ; Accumulate next digit.
or   eax, '0'           ; Convert digit to ASCII.
mov  [edi], al          ; Store digit in memory.
lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-24]
lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[23-0]
cmp  ebx, 1             ; Any nonzero digit yet?
sbb  edi, -1            ; Yes, increment ptr, No, keep old ptr.
shr  eax, 24            ; Next digit
and  edx, 00ffffffh     ; Fraction part
or   ebx, eax           ; Accumulate next digit.
or   eax, '0'           ; Convert digit to ASCII.
mov  [edi], al          ; Store digit in memory.
lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-23]
lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[31-23]
cmp  ebx, 1             ; Any nonzero digit yet?
sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
shr  eax, 23            ; Next digit
and  edx, 007fffffh     ; Fraction part
or   ebx, eax           ; Accumulate next digit.
or   eax, '0'           ; Convert digit to ASCII.
mov  [edi], al          ; Store digit out to memory.
lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-22]
lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[22-0]
cmp  ebx, 1             ; Any nonzero digit yet?
sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
shr  eax, 22            ; Next digit
and  edx, 003fffffh     ; Fraction part
OR   ebx, eax           ; Accumulate next digit.
```

```
      or   eax, '0'           ; Convert digit to ASCII.
      mov  [edi], al          ; Store digit in memory.
      lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-21]
      lea  edx, [edx*4+edx]   ; 5 * fraction, new fraction EDX[21-0]
      cmp  ebx, 1             ; Any nonzero digit yet?
      sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
      shr  eax, 21            ; Next digit
      and  edx, 001fffffh     ; Fraction part
      or   ebx, eax           ; Accumulate next digit.
      or   eax, '0'           ; Convert digit to ASCII.
      mov  [edi], al          ; Store digit in memory.
      lea  eax, [edx*4+edx]   ; 5 * fraction, new digit EAX[31-20]
      cmp  ebx, 1             ; Any nonzero digit yet?
      sbb  edi, -1            ; Yes, increment ptr. No, keep old ptr.
      shr  eax, 20            ; Next digit
      or   eax, '0'           ; Convert digit to ASCII.
      mov  [edi], ax          ; Store last digit and end marker in memory.
      pop  ebx                ; Restore register as per calling convention.
      pop  edi                ; Restore register as per calling convention.
      ret  8                  ; Pop two DWORD arguments and return.
   }
}
```

8.8 Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants

The following examples illustrate the derivation of algorithm, multiplier and shift factor for signed and unsigned integer division.

Unsigned Integer Division

The utility `udiv.exe` was compiled from the code shown in this section. The utilities provided in this document are for reference only and are not supported by AMD.

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 $d = d' * 2n$, the multiplier is the same as for d' and the shift factor is $s + n$.

Example

```
/* This program determines the algorithm (a), multiplier (m), and
    shift factor (s) to be used to accomplish *unsigned* division by
    a constant divisor. Compile with MSVC.
*/

#include <stdio.h>

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 res1, res2;

U32 d, l, s, m, a, r, n, t;
U64 m_low, m_high, j, k;

int main (void)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "Unsigned division by constant\n");
   fprintf(stderr, "=============================\n\n");
```

```
fprintf(stderr, "enter divisor: ");
scanf("%lu", &d);
printf("\n");

if (d == 0) goto printed_code;

if (d >= 0x80000000UL) {
   printf("; dividend: register or memory location\n");
   printf("\n");
   printf("CMP    dividend, 0%08lXh\n", d);
   printf("MOV    EDX, 0\n");
   printf("SBB    EDX, -1\n");
   printf("\n");
   printf("; quotient now in EDX\n");
   goto printed_code;
}

/* Reduce divisor until it becomes odd. */

n = 0;
t = d;
while (!(t & 1)) {
   t >>= 1;
   n++;
}

if (t == 1) {
   if (n == 0) {
      printf("; dividend: register or memory location\n");
      printf("\n");
      printf("MOV    EDX, dividend\n", n);
      printf("\n");
      printf("; quotient now in EDX\n");
   }
   else {
      printf("; dividend: register or memory location\n");
      printf("\n");
      printf("SHR    dividend, %d\n", n);
      printf("\n");
      printf("; quotient replaced dividend\n");
   }
   goto printed_code;
}

/* 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(t) + 1;
```

```
j = (((U64)(0xffffffff)) % ((U64)(t)));
k = (((U64)(1)) << (32 + l)) / ((U64)(0xffffffff - j));
m_low = (((U64)(1)) << (32 + l)) / t;
m_high = (((((U64)(1)) << (32 + l)) + k) / t;
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 and 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(t);
   m_low = (((U64)(1)) << (32 + s)) / ((U64)(t));
   r = ((U32)((((U64)(1)) << (32 + s)) % ((U64)(t))));
   m = (r < ((t >> 1) + 1)) ? ((U32)(m_low)) : ((U32)(m_low)) + 1;
   a = 1;
}

/* Reduce multiplier for either algorithm to smallest possible. */

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

/* Adjust multiplier for reduction of even divisors. */

s += n;

if (a) {
   printf("; dividend: register other than EAX or memory location\n");
   printf("\n");
   printf("MOV    EAX, 0%08lXh\n", m);
   printf("MUL    dividend\n");
   printf("ADD    EAX, 0%08lXh\n", m);
   printf("ADC    EDX, 0\n");
   if (s) printf("SHR    EDX, %d\n", s);
   printf("\n");
   printf("; quotient now in EDX\n");
}
else {
```

```
      printf("; dividend: register other than EAX or memory location\n");
      printf("\n");
      printf("MOV    EAX, 0%08lXh\n", m);
      printf("MUL    dividend\n");
      if (s) printf("SHR    EDX, %d\n", s);
      printf("\n");
      printf("; quotient now in EDX\n");
   }

printed_code:

   fprintf(stderr, "\n");
   exit(0);

   return(0);
}
```

Signed Integer Division

The utility `sdiv.exe` was compiled using the following code. The utilities provided in this document are for reference only and are not supported by AMD.

Example Code

```
/* This program determines the algorithm (a), multiplier (m), and
    shift factor (s) to be used to accomplish *signed* division by
    a constant divisor. Compile with MSVC.
*/

#include <stdio.h>

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);
}

long e;
U32 res1, res2;
U32 oa, os, om;
U32 d, l, s, m, a, r, t;
U64 m_low, m_high, j, k;
```

```
int main(void)

{

   fprintf(stderr, "\n");
   fprintf(stderr, "Signed division by constant\n");
   fprintf(stderr, "============================\n\n");

   fprintf(stderr, "enter divisor: ");
   scanf("%ld", &d);
   fprintf(stderr, "\n");

   e = d;
   d = labs(d);

   if (d == 0) goto printed_code;

   if (e == (-1)) {
      printf("; dividend: register or memory location\n");
      printf("\n");
      printf("NEG    dividend\n");
      printf("\n");
      printf("; quotient replaced dividend\n");
      goto printed_code;
   }
   if (d == 2) {
      printf("; dividend expected in EAX\n");
      printf("\n");
      printf("CMP    EAX, 080000000h\n");
      printf("SBB    EAX, -1\n");
      printf("SAR    EAX, 1\n");
      if (e < 0) printf("NEG    EAX\n");
      printf("\n");
      printf("; quotient now in EAX\n");
      goto printed_code;
   }

   if (!(d & (d - 1))) {
      printf("; dividend expected in EAX\n");
      printf("\n");
      printf("CDQ\n");
      printf("AND    EDX, 0%08lXh\n", (d-1));
      printf("ADD    EAX, EDX\n");
      if (log2(d)) printf("SAR    EAX, %d\n", log2(d));
      if (e < 0)   printf("NEG    EAX\n");
      printf("\n");
      printf("; quotient now in EAX\n");
      goto printed_code;
   }
```

```
   /* Determine algorithm (a), multiplier (m), and shift factor (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;

   if (a) {
      printf("; dividend: memory location or register other than EAX or EDX\n");
      printf("\n");
      printf("MOV    EAX, 0%08LXh\n", m);
      printf("IMUL   dividend\n");
      printf("MOV    EAX, dividend\n");
      printf("ADD    EDX, EAX\n");
      if (s) printf("SAR    EDX, %d\n", s);
      printf("SHR    EAX, 31\n");
      printf("ADD    EDX, EAX\n");
      if (e < 0) printf("NEG    EDX\n");
      printf("\n");
      printf("; quotient now in EDX\n");
   }
   else {
      printf("; dividend: memory location of register other than EAX or EDX\n");
      printf("\n");
      printf("MOV    EAX, 0%08LXh\n", m);
      printf("IMUL   dividend\n");
      printf("MOV    EAX, dividend\n");
      if (s) printf("SAR    EDX, %d\n", s);
      printf("SHR    EAX, 31\n");
      printf("ADD    EDX, EAX\n");
      if (e < 0) printf("NEG    EDX\n");
      printf("\n");
      printf("; quotient now in EDX\n");
   }

printed_code:
```

```
   fprintf(stderr, "\n");
   exit(0);
}
```

Chapter 9 Optimizing with SIMD Instructions

The 64-bit and 128-bit SIMD instructions—SSE and SSE2 instructions—should be used to encode floating-point and integer operation.

- The SIMD instructions use a flat register file rather than the stack register file used by x87 floating-point instructions. This allows arbitrary sequences of operations to map more efficiently to the instruction set.
- Future processors with more or wider multipliers and adders will achieve better throughput using SSE and SSE2 instructions. (Today's processors implement a 128-bit-wide SSE or SSE2 operation as two 64-bit operations that are internally pipelined.)
- SSE and SSE2 instructions work well in both 32-bit and 64-bit threads.

The SIMD instructions provide a theoretical single-precision peak throughput of two additions and two multiplications per clock cycle, whereas x87 instructions can only sustain one addition and one multiplication per clock cycle. The SSE2 and x87 double-precision peak throughput is the same, but SSE2 instructions provide better code density.

This chapter covers the following topics:

9.1 Ensure All Packed Floating-Point Data are Aligned

Optimization

Align all packed floating-point data on 16-byte boundaries.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Misaligned memory accesses reduce the available memory bandwidth and SSE and SSE2 instructions have shorter latencies when operating on aligned memory operands.

Aligning data on 16-byte boundaries allows you to use the aligned load instructions (MOVAPS, MOVAPD, and MOVDQA), which move through the floating-point unit with shorter latencies and reduce the possibility of stalling addition or multiplication instructions that are dependent on the load data.

9.2 Improving Scalar SSE and SSE2 Floating-Point Performance with MOVLPD and MOVLPS When Loading Data from Memory

Optimization

Use the MOVLPS and MOVLPD instructions to move scalar floating-point data into the XMM registers prior to addition, multiplication, or other scalar instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale—Single Precision

The MOVSS instruction is used to move scalar single-precision floating-point data into the XMM registers prior to addition (ADDSS) and multiplication (MULSS) or other scalar instructions. In addition to loading a 32-bit floating-point value into the XMM register, the MOVSS instruction clears the upper 96 bits of the register. Clearing part of the XMM register is an inefficiency that you can bypass by using the MOVLPS instruction. MOVLPS loads two floating-point values from memory without clearing the upper 64 bits of the XMM register.

The latency of the MOVSS instruction is 3 cycles, whereas the latency of the MOVLPS instruction is 2 cycles. The AMD Athlon™ 64 and AMD Opteron™ processors can perform two 64-bit loads per clock cycle. Two 64-bit MOVLPS loads can be issued in the same cycle, assuming the data is 8-byte aligned. Likewise, two MOVSS loads can be performed per cycle, but—unlike MOVLPS—additional operations that interfere with the MULSS and ADDSS instructions must be issued to clear the register. Using MOVLPS rather than MOVSS to load single-precision scalar data from memory on processor-limited floating-point-intensive code can result in significant performance increases.

Consider the following caveats when using the MOVLPS instruction:

- When accessing 4-byte-aligned addresses that are not 8-byte aligned, MOVLPS loads take an additional cycle.
- Since MOVLPS loads two floating-point values instead of one, accessing the last floating-point value in a single-precision array attempts to load 4 bytes of additional memory directly after the end of the array, which may cause an access violation. To avoid an access violation, use MOVSS to access the last value in a single-precision array or store a dummy floating-point value at the end of the array.

- The statement `movlps xmm1, mem64` marks the lower half of XMM1 as FPS (floating-point single-precision) but leaves the upper half of XMM1 unchanged. If XMM1 is later used in any instruction that uses the full 128 bits of XMM1, there can be a performance penalty if the top half is not also in FPS format. Examples of instructions that expect the full 128 bits of XMM1 to be in FPS format are MOVAPS, ANDPS, ANDNPS, and ORPS. For more information on XMM-register data types, see "SSE and SSE2 Instruction and Data Types" on page 351.

Rational—Double Precision

The MOVLPD instruction doesn't necessitate clearing the upper 64 bits of an XMM register, as the MOVSD/MOVQ instructions do, upon loading 64 bits of floating-point data into the lower 64 bits of the XMM register. Using the MOVLPD instruction can significantly increase performance on processor-limited SSE2 scalar floating-point-intensive code.

Consider the following caveat when using the MOVLPD instruction:

- The statement `movlpd xmm1, mem64` marks the lower half of XMM1 as FPD (floating-point double-precision) but leaves the upper half of XMM1 unchanged. If XMM1 is later used in any instruction that uses the full 128 bits of XMM1, there can be a performance penalty if the top half is not also in FPD format. Examples of instructions that expect the full 128 bits of XMM1 to be in FPD format are ANDPD, ANDNPD, and ORPD. For more information on XMM-register data types, see "SSE and SSE2 Instruction and Data Types" on page 351.

9.3 Structuring Code with Prefetch Instructions to Hide Memory Latency

Optimization

When utilizing prefetch instructions, attend to:

- The time allotted (latency) for data to reach the processor between issuing a prefetch instruction and using the data.
- Structuring the code to best take advantage of prefetching.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Prefetch instructions bring the cache line containing a specified memory location into the processor cache. (For more information on prefetch instructions, see “Prefetch Instructions” on page 97.) Prefetching hides the main memory load latency, which is typically many orders of magnitude larger than a processor clock cycle.

There are two types of loops:

Loop type	Description
Memory-limited	Data can be processed and requested faster than it can be fetched from memory.
Processor-limited	Data can be requested and brought into the processor before it is needed because considerable processing occurs during each unrolled loop iteration.

The example provided below illustrates the importance of the above considerations in an example that multiplies a double-precision 32 × 32 matrix **A** with another 32 × 32 transposed double-precision matrix, $\mathbf{B}^{\mathbf{T}}$; the result is returned in another 32 × 32 transposed double-precision matrix, $\mathbf{C}^{\mathbf{T}}$. (The transposition of **B** and **C** is performed to efficiently access their elements because matrices in the C programming language are stored in row-major format. Doing the transposition in advance reduces the problem of matrix multiplication to one of computing several dot-products—one for each element of the results matrix, $\mathbf{C}^{\mathbf{T}}$. This “dotting” operation is implemented as the sum of pair-wise products of the elements of two equal-length vectors.) For this example, assume the processor clock speed is 2 GHz, and the memory latency is 60 ns. In this example, the rows of matrix **A** are repeatedly

"dotted" with a column of $\mathbf{B}^{\mathbf{T}}$. Once this is done, the rows of matrix **A** are "dotted" with the next column of $\mathbf{B}^{\mathbf{T}}$, and the process is repeated through all the columns of $\mathbf{B}^{\mathbf{T}}$.

From a performance standpoint, there are several caveats to recognize, as follows:

- Once all the rows of **A** have been multiplied with the first column of **B**, all the rows of **A** are in the cache, and subsequent accesses to them don't cause cache misses.
- The rows of $\mathbf{B}^{\mathbf{T}}$ are brought into the cache by "dotting" the first four rows of **A** with each row of $\mathbf{B}^{\mathbf{T}}$ in the `Ctr_row_num` for-loop.
- The elements of $\mathbf{C}^{\mathbf{T}}$ are not initially in the cache, and every time a new set of four rows of **A** are "dotted" with a new row of $\mathbf{B}^{\mathbf{T}}$, the processor has to wait for $\mathbf{C}^{\mathbf{T}}$ to arrive in the cache before the results can be written.

You can address the last two caveats by prefetching to improve performance. However, to efficiently exploit prefetching, you must structure the code to issue the prefetch instructions such that:

- Enough time is provided for memory requests sent out through prefetch requests to bring data into the processor's cache before the data is needed.
- The loops containing the prefetch instructions are ordered to issue sufficient prefetch instructions to fetch all the pertinent data.

The matrix order of 32 is not a coincidence. A double-precision number consists of 8 bytes. Prefetch instructions bring memory into the processor in chunks called cache lines consisting of 64 bytes (or eight double-precision numbers). We need to issue four prefetch instructions to prefetch a row of $\mathbf{B}^{\mathbf{T}}$. Consequently, when multiplying all 32 rows of **A** with a particular column of **B**, we want to arrange the for-loop that cycles through the rows of **A** such that it's repeated four times. To achieve this, we need to dot eight rows of **A** with a row of $\mathbf{B}^{\mathbf{T}}$ every time we pass through the `Ctr_row_num` for-loop. Additionally, "dotting" eight rows of **A** upon a row of $\mathbf{B}^{\mathbf{T}}$ produces eight doubles of $\mathbf{C}^{\mathbf{T}}$ (that is, a full cache line).

Assume it takes 60 ns to retrieve data from memory; then we must ensure that at least this much time elapses between issuing the prefetch instruction and the processor loading that data into its registers. The dot-product of eight rows of **A** with a row of $\mathbf{B}^{\mathbf{T}}$ consists of 512 floating-point operations (dotting a single row of **A** with a row of $\mathbf{B}^{\mathbf{T}}$ consists of 32 additions and 32 multiplications). The AMD Athlon, AMD Athlon 64, and AMD Opteron processors are capable of performing a maximum of two floating point operations per clock cycle; therefore, it takes the processor no less than 256 clock cycles to process each `Ctr_row_num` for-loop.

Choosing a matrix order of 32 is convenient for these reasons:

- All three matrices **A**, $\mathbf{B}^{\mathbf{T}}$, and $\mathbf{C}^{\mathbf{T}}$ can fit into the processor's 64-Kbyte L1 data cache.
- On a 2-GHz processor running at full floating-point utilization, 128 ns elapse during the 256 clock cycles, considerably more than the 60 ns to retrieve the data from memory.

- The size of each row is an integer number of cache lines.

A set of eight rows of **A** is dotted in pairs of four with $\mathbf{B}^T$, and prefetches in each iteration of the `Ctr_row_num` for-loop are issued to retrieve:

- The cache line (or set of eight double-precision values) of $\mathbf{C}^T$ to be processed in the next iteration of the `Ctr_row_num` for-loop.
- One quarter of the next row of $\mathbf{B}^T$.

Including the prefetch to the rows of $\mathbf{B}^T$ increases performance by about 16%. Prefetching the elements of $\mathbf{C}^T$ increases performance by an additional 3% or so.

Follow these guidelines when working with processor-limited loops:

- Arrange your code with enough instructions between prefetches so that there is adequate time for the data to be retrieved.
- Make sure the data that you're prefetching fits into the L1 data cache and doesn't displace other data that's also being operated upon. For instance, choosing a larger matrix size might displace **A** if all three matrices can't fit into the 64-Kbyte L1 data cache.
- Operate on data in chunks that are integer multiples of cache lines.

Examples

Double-Precision 32 × 32 Matrix Multiplication

```
//************************************************************************
// This routine multiplies a 32x32 matrix A (stored in row-major format) upon
//  the transpose of a 32x32 matrix B (stored in row-major format) to get
//  the transpose of the resultant 32x32 matrix C.
//************************************************************************
void matrix_multiply_32x32(double *A,double *Btranspose,double *Ctranspose) {
   int Ctr_8col_blck, Ctr_row_num, n;
   // These 4 pointers are used to address 4 consecutive rows of matrix A.
   double *Aptr0, *Aptr1, *Aptr2, *Aptr3;
   // Pointers *Btr_ptr and *Ctr_ptr are used to address the column of B upon
   //  which A is being multiplied and where the result C is placed.
   // Pointers *Bprefptr and *Cprefptr are used to address the next column
   //  of B and the next elements of C to be calculated in advance
   //  using prefetch instructions.
   double *Btr_ptr, *Ctr_ptr, *Btr_prefptr, *Ctr_prefptr;

   // Put the address of matrices B-tranpose and C-transpose into their
   //  respective temporary pointers.
   Btr_ptr = Btranspose; Ctr_ptr = Ctranspose;
   // Shift the prefetch pointers to the next row of B-transpose and the
   //  next set of 8 elements of C-transpose. (Each set of 8 doubles is
   //  a 64-byte cache line if the addresses Btr_ptr and Ctr_ptr are aligned
   //  in memory on 64-byte boundaries.)
```

```
Btr_prefptr = Btr_ptr + 32; Ctr_prefptr = Ctr_ptr + 8;
// This loop cycles through the rows of the TRANSPOSED C matrix. A row
//  of C-transpose is calculated by the code in this loop and then the
//  next row is determined in the following loop iteration. There are
//  32 rows in C-transpose.
for (Ctr_row_num = 0; Ctr_row_num < 32; Ctr_row_num++) {
   // Assign pointers to 4 consecutive rows of A by using the
   //  address of matrix A passed into the function:
   Aptr0 = A;
   Aptr1 = Aptr0 + 32;
   Aptr2 = Aptr0 + 64;
   Aptr3 = Aptr0 + 96;
   // This loop contains code that "dots" 8 rows of A upon the present row
   //  of B-transpose. By looping 4 times, all 32 rows of A are multiplied
   //  upon the present column of B-transpose.
   for (Ctr_8col_blck = 0; Ctr_8col_blck < 4; Ctr_8col_blck++) {
      // This instruction prefetches 1/4 of the next column of B-transpose
      //  upon which matrix A needs to be multiplied. The loop within which
      //  this code resides is executed 4 times, and by incrementing
      //  Btr_prefptr (the ptr to the address of B transpose to be
      //  prefetched) by 8 doubles (or 64 bytes, or 1 cache line) the entire
      //  contents of the next row of B-transpose are brought to the
      //  processor in advance when Ctr_row_num in the outer loop is
      //  incremented
      _mm_prefetch(&Btr_prefptr[0], 2);
      // This loop below "dots" 4 consecutive rows of A upon a row of
      //  B-transpose by looping 8 times through code that multiplies and
      //  accumulates the products of 4 elements of A's rows with 4
      //  elements of B-transpose's column.
      for (n = 0; n < 8; n++) {
         Ctr_ptr[0] += Aptr0[0]*Btr_ptr[0] + Aptr0[1]*Btr_ptr[1] +
                       Aptr0[2]*Btr_ptr[2] + Aptr0[3]*Btr_ptr[3];
         Ctr_ptr[1] += Aptr1[0]*Btr_ptr[0] + Aptr1[1]*Btr_ptr[1] +
                       Aptr1[2]*Btr_ptr[2] + Aptr1[3]*Btr_ptr[3];
         Ctr_ptr[2] += Aptr2[0]*Btr_ptr[0] + Aptr2[1]*Btr_ptr[1] +
                       Aptr2[2]*Btr_ptr[2] + Aptr2[3]*Btr_ptr[3];
         Ctr_ptr[3] += Aptr3[0]*Btr_ptr[0] + Aptr3[1]*Btr_ptr[1] +
                       Aptr3[2]*Btr_ptr[2] + Aptr3[3]*Btr_ptr[3];
         // Increment pointers to B transpose's column and A's rows to
         //  the next 4 elements to be multiplied and accumulated.
         Btr_ptr += 4;
         Aptr0 += 4;
         Aptr1 += 4;
         Aptr2 += 4;
         Aptr3 += 4;
      }
      // The pointer to C-transpose is incremented by 4 doubles to
      //  address the next 4 elements of C-transpose's row to be determined.
      Ctr_ptr += 4;
      // The pointer to B transpose points to the end of the present
      //  row. We need to subtract 32 doubles so Btr_ptr points
```

```
//  again to the top of the column for the next dot-product of
//  4 rows of A upon B-transpose's row vector.
Btr_ptr -= 32;
// The addresses Aptr0, Aptr1, Aptr2, and Aptr3 need to be
//  incremented to the next block of 4 rows of A to be multiplied
//  upon B's column. 4 rows of A are 128 doubles in size, and in
//  the n-loop above they were incremented by 32 already, so they
//  must be incremented an additional 96 to point to the next
//  4 rows of A to be dotted.
Aptr0 += 96;
Aptr1 += 96;
Aptr2 += 96;
Aptr3 += 96;
_mm_prefetch(&Ctr_prefptr[0], 2);
// This loop below "dots" 4 consecutive rows of A upon a row
//  of B-transpose by looping 8 times through code that
//  multiplies and accumulates the products of 4 elements of A's
//  rows with 4 elements of B-transpose's column.
for (n = 0; n < 8; n++) {
   Ctr_ptr[0] += Aptr0[0]*Btr_ptr[0] + Aptr0[1]*Btr_ptr[1] +
                 Aptr0[2]*Btr_ptr[2] + Aptr0[3]*Btr_ptr[3];
   Ctr_ptr[1] += Aptr1[0]*Btr_ptr[0] + Aptr1[1]*Btr_ptr[1] +
                 Aptr1[2]*Btr_ptr[2] + Aptr1[3]*Btr_ptr[3];
   Ctr_ptr[2] += Aptr2[0]*Btr_ptr[0] + Aptr2[1]*Btr_ptr[1] +
                 Aptr2[2]*Btr_ptr[2] + Aptr2[3]*Btr_ptr[3];
   Ctr_ptr[3] += Aptr3[0]*Btr_ptr[0] + Aptr3[1]*Btr_ptr[1] +
                 Aptr3[2]*Btr_ptr[2] + Aptr3[3]*Btr_ptr[3];
   // Increment pointers to B transpose's column and A's rows to
   //  the next 4 elements to be multiplied and accumulated.
   Btr_ptr += 4;
   Aptr0 += 4;
   Aptr1 += 4;
   Aptr2 += 4;
   Aptr3 += 4;
}
// The addresses to prefetch in B-transpose and C-transpose
//  are incremented by 8 doubles, or 64 bytes, or 1 cache line.
//  Each loop of the 4 loops of Ctr_8col_blck above brings in a
//  new set of 8 doubles and after 4 loops the full column of the
//  next column of B and the next set of 8 elements of C to be
//  determined are also brought into the cache.
Btr_prefptr += 8;
Ctr_prefptr += 8;
// The pointer to C-transpose is incremented by 4 doubles
//  to address the next 4 elements of C-transpose's row to be
//  determined.
Ctr_ptr += 4;
// The pointer to B-transpose points to the end of the present
//  row. We need to subtract 32 doubles so Btr_ptr points again
//  to the top of the column for the next dot-product of 4 rows of A
//  upon B-transpose's row vector
```

```
         Btr_ptr -= 32;
         // The addresses Aptr0, Aptr1, Aptr2, and Aptr3 need to be
         //  incremented to the next block of 4 rows of A to be dotted
         //  upon B's column. 4 rows of A are 128 doubles in size, and
         //  in the n-loop above they were incremented by 32 already, so they
         //  must be incremented an additional 96 to point to the
         //  next 4 rows of A to be dotted.
         Aptr0 += 96;
         Aptr1 += 96;
         Aptr2 += 96;
         Aptr3 += 96;
      }
      // Pointer to B-transpose is incremented by a row so as to point
      //  to the next row of B upon which matrix A needs to be multiplied.
      Btr_ptr += 32;
   }
}
```

9.4 Avoid Moving Data Directly Between General-Purpose and MMX™ Registers

Optimization

Avoid moving data directly between general-purpose registers and MMX™ registers; this operation requires the use of the MOVD instruction. If it's absolutely necessary to move data between these two types of registers, use separate store and load instructions to move the data from the source register to a temporary location in memory and then from memory into the destination register, separating the store and the load by at least 10 instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The register-to-register forms of the MOVD instruction are either VectorPath or DirectPath Double instructions. When compared with DirectPath Single instructions, VectorPath and DirectPath Double instructions have comparatively longer execution latencies. In addition, VectorPath instructions prevent the processor from simultaneously decoding other insructions.

Example

Avoid code like this, which copies a value directly from an MMX register to a general-purpose register:

```
movd eax, mm2
```

If it's absolutely necessary to copy a value from an MMX register to a general-purpose register (or vice versa), use separate store and load instructions, separating them by at least 10 instructions:

```
movd DWORD PTR temp, mm2   ; Store the value in memory.
...
; At least 10 other instructions appear here.
...
mov eax, DWORD PTR temp   ; Load the value from memory.
```

9.5 Use MMX™ Instructions to Construct Fast Block-Copy Routines in 32-Bit Mode

Optimization

Use MMX instructions when moving integer data in a block-copy routine.

Application

This optimization applies to:

- 32-bit software

Rationale

MMX instructions relieve the high register pressure typical of x86 code because of the small register file.

In addition, MMX instructions increase the available parallelism on AMD Athlon 64 and AMD Opteron processors because they use both sides (integer and floating-point) of the execution pipeline. For an example of how to move a large quadword-aligned block of data using the MMX MOVQ instruction, see "Optimizing Main Memory Performance for Large Arrays" in the *AMD Athlon™ Processor x86 Code Optimization Guide* (order # 22007).

If a block-copy routine is not used, do not move integer data through MMX registers.

9.6 Passing Data between MMX™ and 3DNow!™ Instructions

Optimization

Avoid passing data between MMX and 3DNow!™ instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rational

The AMD Athlon 64 and AMD Opteron processors do not support bypassing register data between MMX and 3DNow! instructions. One additional cycle of latency is added to a dependency chain whenever data is passed between these instruction groups in either direction.

9.7 Storing Floating-Point Data in MMX™ Registers

Optimization

Avoid storing floating-point data in MMX registers unless using 3DNow! instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using MOVDQ2Q or MOVQ2DQ to shuffle integer data between MMX and XMM registers is useful to relieve register pressure; however, doing so with floating-point data can impact performance. The impact is greater if the floating-point data is denormalized.

9.8 EMMS and FEMMS Usage

Optimization

Use FEMMS or EMMS to clean up the register file between an x87 instruction and a following MMX, 3DNow!, or Enhanced 3DNow! instruction or vice versa.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Use either the FEMMS or the EMMS instruction when switching between the x87 floating-point unit and MMX, 3DNow!, or Enhanced 3DNow! instructions. The FEMMS instruction is aliased to the EMMS instruction on AMD Athlon 64 and AMD Opteron processors. Both instructions convert to an internal NOP instruction in AMD Athlon 64 and AMD Opteron processors. The FEMMS instruction is provided to help ensure that code written for previous generations of AMD processors runs correctly.

There is no penalty for switching between the x87 floating-point instructions and 3DNow! (or MMX) instructions in the processor. The MMX, 3DNow!, and Enhanced 3DNow! instructions are designed to be used concurrently; therefore, no delimiting cleanup operations are required when switching between them. However, x87 and 3DNow!/Enhanced 3DNow!/MMX instructions share the same architectural registers, so there is no easy way to use them concurrently without cleaning up the register file in between by using FEMMS or EMMS. For more information, see *AMD64 Architecture Programmer's Manual Volume 1: Application Programming,* order# 24592.

9.9 Using SIMD Instructions for Fast Square Roots and Fast Reciprocal Square Roots

Optimization

Use SIMD vectorized square root (SQRTPS) and reciprocation (RCCPS) instructions to calculate square roots and reciprocal square roots of single-precision numbers.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

SIMD instructions exist for performing vectorized square root and reciprocation of single-precision numbers. These operations are often used in multimedia applications and also can be utilized in scientific arenas, such as molecular dynamics simulations.

Example

The following function highlights the use of both the vectorized reciprocal and square-root SSE instructions:

```
; reciprocal_sqrt_sse(float *r, float *rcp_sqrt_r, int num_points);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c reciprocal_sqrt_sse.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _reciprocal_sqrt_sse
_reciprocal_sqrt_sse PROC NEAR
;=============================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
;  ENTERED.
; REGISTERS EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   push ebp
   mov  ebp, esp
;=============================================================================
; Parameters passed into routine:
```

```
;  [ebp+8]  = ->r
;  [ebp+12] = ->rcp_sqrt_r
;  [ebp+16] = num_points
;===========================================================================
   push ebx
   push esi
   push edi
;===========================================================================
; THE FIRST 3 ASM LINES BELOW LOAD THE FUNCTION'S ARGUMENTS INTO GENERAL-PURPOSE
;  REGISTERS (GPRS)
;  esi = address of "r"'s to calculate the reciprocal square root of
;  edi = address of "rcp_sqrt_r"'s to store reciprocal square root to
;  ecx = num_points
;===========================================================================
   mov esi,[ebp+8]              ; ESI = ->r
   mov edi,[ebp+12]             ; EDI = ->rcp_sqrt_r
   mov ecx,[ebp+16]             ; ECX = num_points
   mov edx,ecx                  ; EDX = num_points
   mov eax,ecx                  ; EAX = num_points
   shl edx,2                    ; EDX = 4*num_points
   shr eax,4                    ; EAX = num_points/16
   add edi,edx                  ; EDI = -> end of "r"
   add esi,edx                  ; EAX = -> end of "rcp_sqrt_r"
   neg ecx                      ; ECX = -# quadwords of vertices to rotate
   or  eax,eax                  ; If num_points/16 = 0, then skip
                                ;  reciprocal square root.
   jz  skip_recprcl_sqrt_4xloop ; Unroll loop by 4 to work
                                ;  on 16 floats at a time.
;===========================================================================
; THIS LOOP RECIPROCATES AND SQUARE ROOTS 16 FLOATING-POINT NUMBERS EACH
;  LOOP ITERATION AND WORDS WITH THOSE ELEMENTS OF "r" THAT OCCUPY A
;  FULL CACHELINE
;===========================================================================
ALIGN 16                        ; Align address of loop to a 16-byte boundary.
reciprocal_sqrt_4xloop:
   prefetchnta [esi+4*ecx+256]  ; Prefetch the elements "r" 4 cache lines
                                ;  ahead to reciprocate and squareroot 4 loops
                                ;  from now.
   movaps  xmm0, [esi+4*ecx]    ; XMM0=[r3,r2,r1,r0]
   sqrtps  xmm0, xmm0           ; XMM0=[sqrtr3,sqrtr2,sqrtr0,sqrtr0]
   rcpps   xmm0, xmm0           ; XMM0=[1/sqrtr3,1/sqrtr2,1/sqrtr0,1/sqrtr0]
   movaps  xmm1, [esi+4*ecx+16] ; XMM1=[r7,r6,r5,r4]
   sqrtps  xmm1, xmm1           ; XMM1=[sqrtr7,sqrtr6,sqrtr5,sqrtr4]
   rcpps   xmm1, xmm1           ; XMM1=[1/sqrtr7,1/sqrtr6,1/sqrtr5,1/sqrtr4]
   movaps  xmm2, [esi+4*ecx+32] ; XMM2=[r11,r10,r9,r8]
   sqrtps  xmm2, xmm2           ; XMM2=[sqrtr11,sqrtr10,sqrtr9,sqrtr8]
   rcpps   xmm2, xmm2           ; XMM2=[1/sqrtr11,1/sqrtr10,1/sqrtr9,1/sqrtr8]
   movaps  xmm3, [esi+4*ecx+48] ; XMM2=[r15,r14,r13,r12]
   sqrtps  xmm3, xmm3           ; XMM2=[sqrtr15,sqrtr14,sqrtr13,sqrtr12]
   rcpps   xmm3, xmm3           ; XMM2=[1/sqrtr15,1/sqrtr14,1/sqrtr13,1/sqrtr12]
   movntps [edi+4*ecx], xmm0    ; Store reciprocal square root to rcp_sqrt_r.
```

```
   movntps [edi+4*ecx+16], xmm1 ; Store reciprocal square root to rcp_sqrt_r.
   movntps [edi+4*ecx+32], xmm2 ; Store reciprocal square root to rcp_sqrt_r.
   movntps [edi+4*ecx+48], xmm3 ; Store reciprocal square root to rcp_sqrt_r.
   add     ecx, 16              ; Decrement the # of reciprocal square
                                ;  roots to calculate by 16.
   dec     eax                  ; Decrement # of 16 float reciprocal square
                                ;  root loops to perform by 1.
   jnz     reciprocal_sqrt_4xloop
   jmp     skip_recprcl_sqrt_4xloop   ; Jump into loop to calculate reciprocal
                                      ;  square root of floats that don't
                                      ;  occupy a full cache line.

;===========================================================================
; THIS LOOP RECIPROCATES AND SQUARE ROOTS 1 FLOATING POINT NUMBER EACH
;  LOOP ITERATION
;===========================================================================
ALIGN 16                      ; Align address of loop to a 16-byte boundary.
reciprocal_sqrt_1xloop:
   movss  xmm0, [esi+4*ecx]   ; XMM0=[,,,r0]
   sqrtss xmm0, xmm0          ; XMM0=[,,,sqrt(r0)]
   rcpss  xmm0, xmm0          ; XMM0=[,,,1/sqrt(r0)]
   movss  [edi+4*ecx], xmm0   ; Store reciprocal square root to rcp_sqrt_r.
   inc    ecx                 ; Decrement the # of reciprocal square roots
                              ;  to calculate.
skip_recprcl_sqrt_4xloop:
   or     ecx, ecx            ; If ECX != 0, then calculate the reciprocal
                              ;  square root of another float.
   jnz reciprocal_sqrt_1xloop

;===========================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE
;  WAS ENTERED.
; REGISTERS EAX, ECX, AND EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED,
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   pop edi
   pop esi
   pop ebx
   mov esp,ebp
   pop ebp

;===========================================================================
   ret
_reciprocal_sqrt_sse ENDP
_TEXT   ENDS
END
```

The preceding code illustrates the use of separate loops for optimal performance. The loop titled `reciprocal_sqrt_4xloop` works with 16 floating-point numbers in each iteration and is unrolled to keep the processor busy by masking the latencies of the reciprocal and square-root instructions. In general, unrolling loops improves performance by providing opportunities for the processor to work on data pertaining to the next loop iteration while waiting for the result of an operation from the

previous iteration. The `reciprocal_sqrt_1xloop` loop performs the reciprocation and square root on the remaining elements that don't form a full segment of 16 floating-point values. In this chapter, the previous function is the only example that handles any vector stream of `num_points` size. This is done to preserve space, but all examples in this chapter can be modified in a similar manner and used universally.

Additionally, the previous SSE function makes use of the PREFETCHNTA instruction to reduce cache latency. The unrolled loop `reciprocal_sqrt_4xloop` was chosen to work with 64 bytes of data per iteration, which happens to be the size of one cache line (the term used to signify the quantum of data brought into the processor's cache by a memory access, if the data doesn't reside there already). The prefetch causes the processor to load the floating-point operands of the reciprocal and square root operations for the next four loop iterations. While the processor works on the next three iterations, the data for the fourth iteration is sent to the processor. The processor does not have to wait while the aligned SSE instruction MOVAPS is fetched from memory before performing operations on the fourth iteration. This type of memory optimization can be very useful in gaming and high-performance computing, in which data sets are unlikely to reside in the processor's cache. For example, in a simulation involving a million vertices or atoms in which the storage for their coordinates would require 12 bytes per vertex, the total space for the data would be more than 12 Mbytes.

9.10 Use XOR Operations to Negate Operands of SSE, SSE2, and 3DNow!™ Instructions

Optimization

For AMD Athlon, AMD Athlon 64, and AMD Opteron processors, use instructions that perform XOR operations (PXOR, XORPS, and XORPD) instead of multiplication instructions to change the sign bit of operands of SSE , SSE2, and 3DNow! instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

On the AMD Athlon 64 and AMD Opteron processors, using XOR-type instructions allows for more parallelism, as these instructions can execute in either the FADD or FMUL pipe of the floating-point unit.

Single Precision

For single-precision, you can use either 3DNow! or SSE SIMD XOR operations. The latency of multiplying by –1.0 in 3DNow! is 4 cycles, while the latency of using the PXOR instruction is only 2 cycles. Similarly, the latency of the MULPS instruction is 5 cycles, while the latency of the XORPS instruction is 3 cycles. The following code example illustrates how to toggle the sign bit of a number using 3DNow! instructions:

```
signmask DQ 8000000080000000h
pxor mm0, [signmask]   ; Toggle sign bits of both floats.
```

This example does the same thing using SSE instructions:

```
signmask DQ 8000000080000000h,8000000080000000h
xorps xmm0, [signmask]   ; Toggle sign bits of all four floats.
```

Double Precision

To perform double-precision arithmetic, you can use the XORPD instruction—similar to the single-precision example—to flip the sign of packed double-precision floating-point operands. The XORPD instruction takes 3 cycles to execute, whereas the MULPD instruction requires 5 cycles.

```
signmask DQ 8000000000000000h,8000000000000000h
xorpd xmm0, [signmask]   ; Toggle sign bit of both doubles.
```

9.11 Clearing MMX™ and XMM Registers with XOR Instructions

Optimization

Use instructions that perform XOR operations (PXOR, XORPS, and XORPD) to clear all the bits in MMX and XMM registers.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The latency of the MMX XOR instruction (PXOR) is only 3 cycles and comparable to the 3 cycles required to load data, assuming it is in the L1 data cache. The SSE and SSE2 XOR instructions (XORPS and XORPD, respectively) also have latencies of 3 cycles.

Examples

The following examples illustrate how to clear the bits in a register using the different exclusive-OR instructions:

```
; MMX
pxor mm0, mm0   ; Clear the MM0 register.

; SSE
xorps xmm0, xmm0   ; Clear the XMM0 register.

; SSE2
xorpd xmm0, xmm0   ; Clear the XMM0 register.
```

9.12 Finding the Floating-Point Absolute Value of Operands of SSE, SSE2, and 3DNow!™ Instructions

Optimization

Use instructions that perform AND operations (PAND, ANDPS, and ANDPD) to determine the absolute value of floating-point operands of SSE, SSE2, and 3DNow!instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The MMX PAND instruction has a latency of 2 cycles, whereas the SSE and SSE2 AND instructions (ANDPS and ANDPD, respectively) have latencies of 3 cycles. The following examples illustrate how to clear the sign bits:

```
; 3DNow!
absmask DQ 7FFFFFFF7FFFFFFFh
pand mm0, [absmask]   ; Clear the sign bits of both floats in MM0.

; SSE
absmask DQ 7FFFFFFF7FFFFFFFh,7FFFFFFF7FFFFFFFh
andps xmm0, [absmask]   ; Clear the sign bits of all four floats in XMM0.

; SSE2
absmask DQ 7FFFFFFFFFFFFFFFh,7FFFFFFFFFFFFFFFh
andpd xmm0, [absmask]   ; Clear the sign bits of both doubles in XMM0.
```

9.13 Accumulating Single-Precision Floating-Point Numbers Using SSE, SSE2, and 3DNow!™ Instructions

Optimization

In 32-bit software, use the 3DNow! PFACC instruction to perform complex-number multiplication, 4 × 4 matrix multiplication, and dot products. For 64-bit software, careful selection of SSE instructions based on how the data is organized can also lead to more efficient code, as shown in the second example.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Though SSE, SSE2, and 3DNow! instrucitons are similar in the sense that they all have vectorized multiplication and addition, 3DNow! technology supports certain special instructions. One of these is the PFACC instruction. There are many instances where PFACC is useful, such as complex-number multiplication, 4 × 4 matrix multiplication, and dot products.

Examples

The following example accumulates two floats in two MMX registers:

```
;accumulate_3dnow(float *a_and_b, float *c_and_d, float *aplusb_cplusd);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c accumulate_3dnow.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _accumulate_3dnow
_accumulate_3dnow PROC NEAR

;============================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS ENTERED
; REGISTERS (EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED)
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
```

```
push ebp
mov  ebp, esp
;===========================================================================
; Parameters passed into routine:
;   [ebp+8]  = ->a_and_b
;   [ebp+12] = ->c_and_d
   ;   [ebp+16] = ->aplusb_cplusd
;===========================================================================
push ebx
push esi
push edi
;===========================================================================
; THE 4 ASM LINES BELOW LOAD THE FUNCTION'S ARGUMENTS INTO GENERAL-PURPOSE
;  REGISTERS (GPRS)
;  esi = starting address of 2 floats "a_and_b"
;  edi = starting address of 2 floats "c_and_d"
;  eax = starting address of 2 floats "aplusb_cplusd"
;===========================================================================
mov esi, [ebp+8]    ; esi = ->a_and_b
mov edi, [ebp+12]   ; edi = ->c_and_d
mov eax, [ebp+16]   ; eax = ->aplusb_cplusd
;===========================================================================
; ADD a AND b TOGETHER AND ALSO c AND d
;===========================================================================
emms
movq  mm0, [esi]   ; mm0 = [b,a]
movq  mm1, [edi]   ; mm1 = [d,c]
pfacc mm0, mm1     ; mm0 = [c+d,b+a]
;===========================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE
;  WAS ENTERED
; REGISTERS (EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED)
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
pop edi
pop esi
pop ebx
mov esp,ebp
pop ebp
;===========================================================================
ret
_accumulate_3dnow ENDP
_TEXT   ENDS
END
```

The same operation can be performed using SSE instructions, but the data in the XMM registers must be rearranged. The next example loads four floating-point values into four XMM registers, XMM4–XMM7, and then rearranges and adds the values so as to accumulate the sum of each XMM register into a float in XMM1.

```
;-----------------------------------------------------------------
; The instructions below take the 4 floats in each XMM register below:
```

```
;  xmm4  =  [d,c,b,a]
;  xmm5  =  [D,C,B,A]
;  xmm6  =  [h,g,f,e]
;  xmm7  =  [H,G,F,E]
;
;  and arranges them to look like:
;  xmm4  =  [E,e,A,a]
;  xmm1  =  [F,f,B,b]
;  xmm2  =  [G,g,C,c]
;  xmm3  =  [H,h,D,d]

movaps   xmm3, xmm4   ; xmm3 | [d,c,b,a]
movaps   xmm0, xmm5   ; xmm0 | [D,C,B,A]

unpcklps xmm4, xmm6   ; xmm4 | [f,b,e,a]
unpckhps xmm3, xmm6   ; xmm3 | [h,d,g,c]
movaps   xmm1, xmm4   ; xmm1 | [f,b,e,a]
movaps   xmm2, xmm3   ; xmm2 | [h,d,g,c]

unpcklps xmm5, xmm7   ; xmm5 | [F,B,E,A]
unpckhps xmm0, xmm7   ; xmm0 | [H,D,G,C]

unpcklps xmm4, xmm5   ; xmm4 | [E,e,A,a]
unpckhps xmm1, xmm5   ; xmm1 | [F,f,B,b]
unpcklps xmm3, xmm0   ; xmm3 | [G,g,C,c]
unpckhps xmm2, xmm0   ; xmm2 | [H,h,D,d]

; Now if we compute the sum of these registers, we get the dot-product
;  of the first row of A with vector X:
;
;  a+b+c+d
;
;  in the lower DWORD of the resultant XMM register. The dot-product of the
;  second row is stored in the second DWORD and so on, such that:
;
;  xmm1 = [V+X+Y+Z,v+x+y+z,A+B+C+D,a+b+c+d]

addps xmm1, xmm4   ; xmm1 | [E+F,e+f,A+B,a+b]
addps xmm3, xmm2   ; xmm3 | [G+H,g+h,C+D,c+d]
addps xmm1, xmm3   ; xmm1 | [E+F+G+H,e+f+g+h,A+B+C+D,a+b+c+d]
```

9.14 Complex-Number Arithmetic Using SSE, SSE2, and 3DNow!™ Instructions

Optimization

Use vectorizing SSE, SSE2 and 3DNow! instructions to perform complex number calculations.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Complex numbers have a "real" part and an "imaginary" part (where the imaginary part is denoted by the letter *i*). For example, the complex number *z1* might have a real part equal to 4 and an imaginary part equal to 3, written as 4 + 3i. Multiplying and adding complex numbers is an integral part of digital signal processing. Complex number addition is illustrated here using two complex numbers, *z1* (4 + 3i) and *z2* (5 + 2i):

z1 + *z2* = (4 + 3i) + (5 + 2i) = [4+5] + [3+2]i = 9 + 5i

or:

```
sum.real = z1.real + z2.real
sum.imag = z1.imag + z2.imag
```

Complex number addition is illustrated here using the same two complex numbers:

z1 + *z2* = (4 + 3i)(5 + 2i) = [4 × 5 - 3 × 2] + [3 × 5 + 4 × 2]i = 14 + 23i

or:

```
product.real = z1.real * z2.real - z1.imag * z2.imag
product.imag = z1.real * z2.imag + z1.imag * z2.real
```

Complex numbers are stored as streams of two-element vectors, the two elements being the real and imaginary parts of the complex numbers. Addition of complex numbers can be achieved using vectorizing SSE or 3DNow!instructions, such as PFADD, ADDPS, and ADDPD. Multiplication of complex numbers is more involved.

From the formulas for multiplication, the real and imaginary parts of one of the numbers needs to be interchanged, and, additionally, the products must be positively or negatively accumulated depending upon whether we are computing the imaginary or real portion of the product.

The following functions use SSE and 3DNow! instructions to illustrate complex multiplication of streams of complex numbers `x[]` and `y[]` stored in a product stream `prod[]`. For these examples, assume that the sizes of `x[]` and `y[]` are even multiples of four.

Examples

Listing 25. Complex Multiplication of Streams of Complex Numbers (SSE)

```
; cmplx_multiply_sse(float *x, float *y, int num_cmplx_elem, float *prod);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c cmplx_multiply_sse.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _cmplx_multiply_sse
_cmplx_multiply_sse PROC NEAR
;===============================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS ENTERED
; REGISTERS (EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED)
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   push ebp
   mov  ebp, esp
;===============================================================================
; parameters passed into routine:
;   [ebp+8]  = ->x
;   [ebp+12] = ->y
;   [ebp+16] = num_cmplx_elem
;   [ebp+20] = ->prod
;===============================================================================
   push ebx                    ;   preserve contents in ebx,esi, and edi on stack
   push esi                    ;
   push edi                    ;
;===============================================================================
; THE CODE BELOW PUTS THE FLOATING POINT SIGN MASK
;  [80000000000000008000000000000000h]
;  TO FLIP THE SIGN OF PACKED SINGLE PRECISION NUMBERS BY USING XORPS
;===============================================================================
   mov eax, esp              ; Copy stack pointer into EAX.
   mov ebx, 16
   sub esp, 32               ; Subtract 32 bytes from stack pointer.
   and eax, 15               ; AND old stack pointer address with 15 to
                             ;  determine # of bytes the address is past a
                             ;  16-byte-aligned address.
   sub ebx, eax              ; EBX = # of bytes above ESP to next
                             ;  16-byte-aligned address
   mov edi, 0h               ; EDI = 00000000h
   mov esi, 80000000h        ; EBX = 80000000h
   shr ebx, 2                ; EBX = # of DWORDs past 16-byte-aligned address
```

```
   mov [esp+4*ebx+12], esi   ; Move into address esp+4*ebx the single-precision
   mov [esp+4*ebx+8], edi    ;  floating-point sign mask.
   mov [esp+4*ebx+4], esi
   mov [esp+4*ebx], edi
;===========================================================================
; THE 4 ASM LINES BELOW LOAD THE FUNCTION's ARGUMENTS INTO GENERAL-PURPOSE
;  REGISTERS (GPRS)
;       esi = address of array "x"
;       edi = address of array "y"
;       ecx = # of cmplx products to compute
;       eax = address of product to which results are stored
;===========================================================================
    mov esi, [ebp+8]    ; esi = ->x
    mov edi, [ebp+12]   ; edi = ->y
    mov ecx, [ebp+16]   ; ecx = num_cmplx_elem
    mov eax, [ebp+20]   ; eax = ->prod
;===========================================================================
; THE 6 ASM LINES BELOW OFFSET THE ADDRESS TO THE ARRAYS x[] AND y[] SUCH
;  THAT THEY CAN BE ACCESSED IN THE MOST EFFICIENT MANNER AS ILLUSTRATED
;  BELOW IN THE LOOP mult4cmplxnum_loop WITH THE MINIMUM NUMBER OF
;  ADDRESS INCREMENTS
;===========================================================================
   mov edx, ecx   ;   edx = num_cmplx_elem
   neg ecx        ;   ecx = -num_cmplx_elem
   shl edx, 3     ; edx = 8 * num_cmplx_elem = # bytes in x[] and y[] to multiply
   add esi, edx   ; esi = -> to last element of x[] to multiply
   add edi, edx   ; edi = -> to last element of y[] to multiply
   add eax, edx   ; eax = -> end of prod[] to calculate
;===========================================================================
; THIS LOOP MULTIPLIES 4 COMPLEX #s FROM "x[]" UPON 4 COMPLEX #s FROM "y[]"
;  AND RETURNS THE PRODUCT IN "prod[]".
;===========================================================================
ALIGN 16                    ; Align address of loop to a 16-byte boundary.
eight_cmplx_prod_loop:
   movaps xmm0, [esi+ecx*8]       ; xmm0=[x1i,x1r,x0i,x0r]
   movaps xmm1, [esi+ecx*8+16]    ; xmm1=[x3i,x3r,x2i,x2r]
   movaps xmm4, [edi+ecx*8]       ; xmm4=[y1i,y1r,y0i,y0r]
   movaps xmm5, [edi+ecx*8+16]    ; xmm5=[y3i,y3r,y2i,y2r]
   movaps xmm2, xmm0              ; xmm2=[x1i,x1r,x0i,x0r]
   movaps xmm3, xmm1              ; xmm3=[x3i,x3r,x2i,x2r]
   movaps xmm6, xmm4              ; xmm6=[y1i,y1r,y0i,y0r]
   movaps xmm7, xmm5              ; xmm7=[y3i,y3r,y2i,y2r]
   shufps xmm0, xmm0, 10100000b   ; xmm0=[x1r,x1r,x0r,x0r]
   shufps xmm1, xmm1, 10100000b   ; xmm1=[x3r,x3r,x2r,x2r]
   shufps xmm2, xmm2, 11110101b   ; xmm2=[x1i,x1i,x0i,x0i]
   shufps xmm3, xmm3, 11110101b   ; xmm3=[x3i,x3i,x2i,x2i]
   xorps  xmm6, [esp+4*ebx]       ; xmm6=[-y1i,y1r,-y0i,y0r]
   xorps  xmm7, [esp+4*ebx]       ; xmm7=[-y3i,y3r,-y2i,y2r]
   mulps xmm0, xmm4               ; xmm0=[x1r*y1i,x1r*y1r,x0r*y0i,x0r*y0r]
   mulps xmm1, xmm5               ; xmm1=[x3r*y3i,x3r*y3r,x2r*y2i,x2r*y2r]
   shufps xmm7, xmm7, 10110001b   ; xmm7=[y3r,-y3i,y2r,-y2i]
```

```
   mulps xmm2, xmm6              ; xmm2=[x1i*y1r,-x1i*y1i,x0i*y0r,-x0i*y0i]
   mulps xmm3, xmm7              ; xmm3=[x3i*y3r,-x3i*y3i,x2i*y2r,-x2i*y2i]
   addps xmm0, xmm2              ; xmm0=[x1r*y1i+x1i*y1r,x1r*y1r-x1i*y1i,
                                 ;  x0r*y0i+x0i*y0r,x0r*y0r-x0i*y0i]
   addps xmm1, xmm3              ; xmm1=[x3r*y3i+x3i*y3r,x3r*y3r-x3i*y3i,
                                 ;  x2r*y2i+x2i*y2r,x2r*y2r-x2i*y2i]
   movntps [eax+ecx*8], xmm0     ; Stream XMM0 and XMM1 to representative
   movntps [eax+ecx*8+16], xmm1  ;  memory address of prod[].
   add     ecx, 4                ; ECX = ECX + 4
   jnz     eight_cmplx_prod_loop
;===========================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
;  ENTERED
; REGISTERS EAX, ECX, AND EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   add esp, 32
   pop edi
   pop esi
   pop ebx
   mov esp, ebp
   pop ebp
;===========================================================================
   ret
_cmplx_multiply_sse ENDP
_TEXT   ENDS
END
```

Listing 26. Complex Multiplication of Streams of Complex Numbers (3DNow!™ Technology)

```
; cmplx_multiply_3dnow(float *x, float *y, int num_cmplx_elem, float *prod);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c cmplx_multiply_3dnow.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _cmplx_multiply_3dnow
;cmplx_multiply_3dnow(float *x, float *y, int num_cmplx_elem, float *prod);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c cmplx_multiply_3dnow.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _cmplx_multiply_3dnow
_cmplx_multiply_3dnow PROC NEAR
;===========================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS ENTERED
```

```
; REGISTERS EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   push ebp
   mov  ebp, esp
;===========================================================================
; Parameters passed into routine:
;  [ebp+8]  = ->x
;  [ebp+12] = ->y
;  [ebp+16] = num_cmplx_elem
;  [ebp+20] = ->prod
;===========================================================================
   push ebx
   push esi
   push edi
;===========================================================================
; THE 4 ASM LINES BELOW LOAD THE FUNCTION's ARGUMENTS INTO GENERAL-PURPOSE
;  REGISTERS (GPRS)
;  esi = address of array "x"
;  edi = address of array "y"
;  ecx = # of cmplx products to compute
;  eax = address of product to which results are stored
;===========================================================================
   mov esi, [ebp+8]    ; esi = ->x
   mov edi, [ebp+12]   ; edi = ->y
   mov ecx, [ebp+16]   ; ecx = num_cmplx_elem
   mov eax, [ebp+20]   ; eax = ->prod
;===========================================================================
; THE 6 ASM LINES BELOW OFFSET THE ADDRESS TO THE ARRAYS x[] AND y[] SUCH
;  THAT THEY CAN BE ACCESSED IN THE MOST EFFICIENT MANNER AS ILLUSTRATED
;  BELOW IN THE LOOP mult4cmplxnum_loop WITH THE MINIMUM NUMBER OF
;  ADDRESS INCREMENTS
;===========================================================================
   mov  edx, ecx   ; edx = num_cmplx_elem]
   neg  ecx        ; ecx = -num_cmplx_elem
   imul edx, 8     ; edx = 8 * num_cmplx_elem = # bytes in x[] and y[] to multiply
   add  esi, edx   ; esi = -> to last element of x[] to multiply
   add  edi, edx   ; edi = -> to last element of y[] to multiply
   add  eax, edx   ; eax = -> end of prod[] to calculate
;===========================================================================
; THIS LOOP MULTIPLIES 4 COMPLEX #s FROM "x[]" UPON 4 COMPLEX #s FROM "y[]"
;  AND RETURNS THE PRODUCT IN "prod[]".
;===========================================================================
ALIGN 16                    ; Align address of loop to a 16-byte boundary.
four_cmplx_prod_loop:                          ;
   movq    mm0, QWORD PTR [esi+ecx*8]      ; mm0=[x0i,x0r]
   movq    mm1, QWORD PTR [esi+ecx*8+8]    ; mm1=[x1i,x1r]
   movq    mm2, QWORD PTR [esi+ecx*8+16]   ; mm2=[x2i,x2r]
   movq    mm3, QWORD PTR [esi+ecx*8+24]   ; mm3=[x3i,x3r]
   pswapd  mm4, QWORD PTR [esi+ecx*8]      ; mm4=[x0r,x0i]
   pswapd  mm5, QWORD PTR [esi+ecx*8+8]    ; mm5=[x1r,x1i]
   pswapd  mm6, QWORD PTR [esi+ecx*8+16]   ; mm6=[x2r,x2i]
```

```
   pswapd  mm7, QWORD PTR [esi+ecx*8+24]   ; mm7=[x3r,x3i]
   pfmul   mm0, QWORD PTR [edi+ecx*8]      ; mm0=[x0i*y0i,x0r*y0r]
   pfmul   mm1, QWORD PTR [edi+ecx*8+8]    ; mm1=[x1i*y1i,x1r*y1r]
   pfmul   mm2, QWORD PTR [edi+ecx*8+16]   ; mm2=[x2i*y2i,x2r*y2r]
   pfmul   mm3, QWORD PTR [edi+ecx*8+24]   ; mm3=[x3i*y3i,x3r*y3r]
   pfmul   mm4, QWORD PTR [edi+ecx*8]      ; mm4=[x0r*y0i,x0i*y0r]
   pfmul   mm5, QWORD PTR [edi+ecx*8+8]    ; mm5=[x1r*y1i,x1i*y1r]
   pfmul   mm6, QWORD PTR [edi+ecx*8+16]   ; mm6=[x2r*y2i,x2i*y2r]
   pfmul   mm7, QWORD PTR [edi+ecx*8+24]   ; mm7=[x3r*y3i,x3i*y3r]
   pfpnacc mm0, mm4              ; mm0=[x0r*y0i+x0i*y0r,x0r*y0r-x0i*y0i]
   pfpnacc mm1, mm5              ; mm1=[x1r*y1i+x1i*y1r,x1r*y1r-x1i*y1i]
   pfpnacc mm2, mm6              ; mm2=[x2r*y2i+x2i*y2r,x2r*y2r-x2i*y2i]
   pfpnacc mm3, mm7              ; mm3=[x3r*y3i+x3i*y3r,x3r*y3r-x3i*y3i]
   movntq  [eax+ecx*8], mm0      ; Stream MM0-MM3 to representative memory
   movntq  [eax+ecx*8+8], mm1    ;  addresses of prod[]
   movntq  [eax+ecx*8+16], mm2
   movntq  [eax+ecx*8+24], mm3
   add     ecx, 4                ; ECX = ECX + 4
   jnz     four_cmplx_prod_loop
;===========================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
;  ENTERED
; REGISTERS EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   femms
   pop edi
   pop esi
   pop ebx
   mov esp, ebp
   pop ebp
;===========================================================================
   ret
_cmplx_multiply_3dnow ENDP
_TEXT   ENDS
END
```

The illustrations above make use of many optimization techniques. First, the 3DNow! technology code utilizes the PSWAPD and PFPNACC instructions, whose operations are outlined below:

```
; PSWAPD
; Suppose that MM0 contains two floats: r and i.
; INPUT:
;  MM0 = [i,r]
; OUTPUT:
;  MM1 = [r,i]

pswapd mm1, mm0   ; MM1 = [r,i]

; Additionally, PSWAPD can be used with a 64-bit memory location. Suppose
;  that EDI contains the address of two floats: r and i.
; INPUT:
```

```
;  [EDI:EDI+8] = [b,a]
; OUTPUT:
;  MM1 = [r,i]

pswapd mm1, [edi]   ; MM1 = [r,i]


; PFPNACC
; Suppose that MM0 contains two floats: r1 * r2 (the product of the real parts
;  of 2 complex numbers) and i1 * i2 (the product of the imaginary parts
;  of 2 complex numbers).
;  Also suppose that MM1 contains two floats: r1 * i2 (the product of the real
;  part of the first complex number and the imaginary part of the second
;  complex number) and i1 * r2 (the product of the imaginary part of the
;  first complex number and the real part of the second complex number).
; INPUTS:
;  MM0 = [i1*i2,r1*r2]
;  MM1 = [i1*r2,r1*i2]
; OUTPUT:
;  MM0 = [r1*i2+i1*r2,r1*r2-i1*i2]

pfpnacc mm0, mm1   ; MM0 = [r1*i2+i1*r2,r1*r2-i1*i2]

; Additionally, PSWAPD can be used with a 64-bit memory location. Suppose
;  that EDI contains the address of two floats: r1 * i2 (the product of the
;  real part of the first complex number and the imaginary part of the
;  second complex number) and i1 * r2 (the product of the imaginary part of
;  the first complex and the real part of the second complex number).
; INPUTS:
;  MM0 = [i1*i2,r1*r2]
;  [EDI:EDI+8] = [i1*r2,r1*i2]
; OUTPUT:
;  MM0 = [r1*i2+i1*r2,r1*r2-i1*i2]

pfpnacc mm0, [edi]   ; MM0 = [r1*i2+i1*r2,r1*r2-i1*i2]
```

The PFPNACC instruction is specifically designed for use in complex arithmetic operations.

Additionally, four complex numbers are concurrently multiplied in the examples using SSE and 3DNow! instructions to break up register dependencies. Loads, multiplications, and additions do not execute with zero delay, but have a latency associated with them. The following instructions:

```
movq    mm0, QWORD PTR [esi+ecx*8]  ; mm0 = [x0i,x0r]
pswapd  mm4, QWORD PTR [esi+ecx*8]  ; mm4 = [x0r,x0i]
pfmul   mm0, QWORD PTR [edi+ecx*8]  ; mm0 = [x0i*y0i,x0r*y0r]
pfmul   mm4, QWORD PTR [edi+ecx*8]  ; mm4 = [x0r*y0i,x0i*y0r]
pfpnacc mm0, mm4                    ; mm0 = [x0r*y0i+x0i*y0r,x0r*y0r-x0i*y0i]
```

are dependent upon one another. The move from memory (MOVQ) requires 2 cycles, PSWAPD also requires 2 cycles, the two PFMUL instructions require 6 cycles, and PFPNACC requires 6 cycles. The instruction flow through the processor is illustrated on a clock-cycle basis, as follows:

```
Instruction     0     2     4     6     8     10    12    14

MOVQ            xxxxxx
PSWAPD          xxxxxx
PFMUL                xxxxxxxxxxxxxxxxxx
PFMUL                   xxxxxxxxxxxxxxxxxx
PFPNACC                                    xxxxxxxxxxxxxxxxxx
```

and takes 15 cycles to finish. During this 15 cycles, the processor has the ability to perform 60 single-precision floating-point operations, of which it only performs six. The majority of the time is spent waiting for previous instructions to terminate so that arguments to future instructions are available. By unrolling the multiplication, working with four complex numbers per clock, there are enough instructions that aren't dependent on previous or presently executing operations so that the processor can mask the execution latency by keeping itself busy, as illustrated below:

```
Instruction     0     2     4     6     8     10    12    14    16    18

MOVQ            xxxxxx
MOVQ            xxxxxx
MOVQ               xxxxxx
MOVQ               xxxxxx
PSWAPD                xxxxxx
PSWAPD                xxxxxx
PSWAPD                   xxxxxx
PSWAPD                   xxxxxx
PFMUL                 xxxxxxxxxxxxxxxxxx
PFMUL                    xxxxxxxxxxxxxxxxxx
PFMUL                       xxxxxxxxxxxxxxxxxx
PFMUL                          xxxxxxxxxxxxxxxxxx
PFMUL                             xxxxxxxxxxxxxxxxxx
PFMUL                                xxxxxxxxxxxxxxxxxx
PFMUL                                   xxxxxxxxxxxxxxxxxx
PFPNACC                              xxxxxxxxxxxxxxxxxx
PFPNACC                                 xxxxxxxxxxxxxxxxxx
PFPNACC                                    xxxxxxxxxxxxxxxxxx
PFPNACC                                       xxxxxxxxxxxxxxxxxx
```

Multiplying four complex single-precision numbers only takes 17 cycles as opposed to 14 cycles to multiply one complex single-precision number. The floating-point pipes are kept busy by feeding new instructions into the floating-point pipeline each cycle. In the arrangement above, 24 floating-point operations are performed in 17 cycles, achieving more than a 3.5x increase in performance.

The last optimization in both implementations is the use of the MOVNTQ and MOVNTPS instructions, nontemporal writes to memory that stream data to main memory. These instructions increase throughput to memory and make more efficient use of the bandwidth provided by the

processor and memory controller. Nontemporal writes, such as MOVNTQ, MOVNTPS, and MOVNTDQ, should only be used on data that is not going to be accessed again in the near future.

9.15 Optimized 4 × 4 Matrix Multiplication on 4 × 1 Column Vector Routines

Optimization

Transpose the rotation matrix to eliminate the need to accumulate floating-point values in an XMM register.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The multiplication of a 4 × 4 matrix with a 4 × 1 vector is commonly used in 3-D graphics for geometric transformation (translating, scaling, rotating, and applying perspective to 3-D points represented in homogeneous coordinates). Efficiency in single-precision matrix multiplication can be enhanced by use of SIMD instructions to increase throughput, but there are other general optimizations that can be implemented to further increase performance. The first optimization is the transposition of the rotation matrix such that the column *n* of the matrix becomes the row *n* and the row *m* becomes the column *m*. This optimization doesn't benefit 3DNow! technology code (3DNow! technology has extended instructions that preclude the need for this optimization), but does benefit SSE code. There are no SSE or SSE2 instructions that accumulate the floats and doubles in a single XMM register; for this reason, the matrix must be transposed. If the rotation matrix isn't transposed, then the dot-product of a row of the matrix with a column vector necessitates the accumulation of the four floating-point values in an XMM register. The multiplication upon the column vector is illustrated here:

```
                |r00 r01 r02 r03|       |r00 r10 r20 r30|   |v0|   |v'0|
tr(R) x v = tr |r10 r11 r12 r13| x v = |r01 r11 r21 r31| x |v1| = |v'1|
                |r20 r21 r22 r23|       |r02 r12 r22 r32|   |v2|   |v'2|
                |r30 r31 r32 r33|       |r03 r13 r23 r33|   |v3|   |v'3|

          Step 0       Step 1       Step 2       Step 3
|v'0|   |r00 x v0|   |r01 x v1| + |r02 x v2| + |r03 x v3|
|v'1| = |r10 x v0| + |r11 x v1| + |r12 x v2| + |r13 x v3|
|v'2|   |r20 x v0|   |r21 x v1| + |r22 x v2| + |r23 x v3|
|v'3|   |r30 x v0|   |r31 x v1| + |r32 x v2| + |r33 x v3|
```

In each step above, the elements of the rotation matrix can be loaded into an XMM register with the MOVAPS instruction, assuming the rotation matrix begins at a 16-byte-aligned memory location. Transposition of the rotation matrix eliminates the need to accumulate the floating-point values in an

XMM register, but it does require the duplication of the elements of the 4×1 column vector **V** in all four floating-point values of the XMM register in each step above. Listing 27 is an SSE function that performs 4×4 matrix multiplication upon a stream of `num_vertices_to_rotate` vertices.

Examples

Listing 27. 4×4 Matrix Multiplication (SSE)

```
; matrix_x_vector_sse(float *trR, float *v, int num_vertices_to_rotate,
   float *rotv);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c matrix_x_vector_sse.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _matrix_x_vector_sse
_matrix_x_vector_sse PROC NEAR
;=============================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
;  ENTERED.
; REGISTERS EAX, ECX, AND EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED,
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   push ebp
   mov  ebp, esp
;=============================================================================
; Parameters passed into routine:
;  [ebp+8]  = ->trR
;  [ebp+12] = ->v
;  [ebp+16] = num_vertices_to_rotate
;  [ebp+20] = ->rotv
;=============================================================================
   push ebx
   push esi
   push edi
;=============================================================================
; THE 4 ASM LINES BELOW LOAD THE FUNCTION's ARGUMENTS INTO GENERAL-PURPOSE
;  REGISTERS (GPRS)
;  esi = address of Transposed Rotation Matrix
;  edi = address of vertices to rotate
;  ecx = # of vertices to rotate
;  eax = address of rotated vertices
;=============================================================================
   mov  esi, [ebp+8]    ; ESI = ->trR
   mov  edi, [ebp+12]   ; EDI = ->v
   mov  ecx, [ebp+16]   ; ECX = num_vertices_to_rotate
   mov  edx, ecx        ; EDX = num_vertices_to_rotate
   shl  edx, 4          ; EDX = 16*num_vertices_to_rotate
   mov  eax, [ebp+20]   ; EAX = ->rotv
```

```
   imul ecx, 2          ; ECX = # quadwords of vertices to rotate
   add  edi, edx        ; EDI = -> end of "v"
   add  eax, edx        ; EAX = -> end of "rotv"
   neg  ecx             ; ECX = -# quadwords of vertices to rotate
;============================================================================
; THE 4 ASM LINES BELOW LOAD THE TRANSPOSED ROTATION MATRIX "R" INTO XMM0-XMM3
;  IN THE FOLLOWING MANNER:
;  xmm0 = column 0 of "R" or row 0 of "R" transpose
;  xmm1 = column 1 of "R" or row 1 of "R" transpose
;  xmm2 = column 2 of "R" or row 2 of "R" transpose
;  xmm3 = column 3 of "R" or row 3 of "R" transpose
;============================================================================
   movaps xmm0, [esi]      ; XMM0 = [R30,R20,R10,R00]
   movaps xmm1, [esi+16]   ; XMM1 = [R31,R21,R11,R01]
   movaps xmm2, [esi+32]   ; XMM2 = [R32,R22,R12,R02]
   movaps xmm3, [esi+48]   ; XMM3 = [R33,R23,R13,R03]
;============================================================================
; THIS LOOP ROTATES "num_vertices_to_rotate" VERTICES BY THE TRANSPOSED
;  ROTATION MATRIX "R" PASSED INTO THE ROUTINE AND STORES THE ROTATED
;  VERTICES TO "rotv".
;============================================================================
ALIGN 16                        ; Align address of loop to a 16-byte boundary.
rotate_vertices_loop:
   movlps   xmm4, [edi+8*ecx]     ; XMM4=[,,v1,v0]
   movlps   xmm6, [edi+8*ecx+8]   ; XMM6=[,,v3,v2]
   unpcklps xmm4, xmm4            ; XMM4=[v1,v1,v0,v0]
   unpcklps xmm6, xmm6            ; XMM6=[v3,v3,v2,v2]
   movhlps  xmm5, xmm4            ; XMM5=[,,v1,v1]
   movhlps  xmm7, xmm6            ; XMM7=[,,v3,v3]
   movlhps  xmm4, xmm4            ; XMM4=[v0,v0,v0,v0]
   mulps    xmm4, xmm0            ; XMM4=[R30*v0,R20*v0,R10*v0,R00*v0]
   movlhps  xmm5, xmm5            ; XMM5=[v1,v1,v1,v1]
   mulps    xmm5, xmm1            ; XMM5=[R31*v1,R21*v1,R11*v1,R01*v1]
   movlhps  xmm6, xmm6            ; XMM6=[v2,v2,v2,v2]
   mulps    xmm6, xmm2            ; XMM6=[R32*v2,R22*v2,R12*v2,R02*v2]
   addps    xmm4, xmm5            ; XMM4=[R30*v0+R31*v1,R20*v0+R21*v1,
                                  ;  R10*v0+R11*v1,R00*v0+R01*v1]
   movlhps  xmm7, xmm7            ; XMM7=[v3,v3,v3,v3]
   mulps    xmm7, xmm3            ; XMM6=[R33*v3,R23*v3,R13*v3,R03*v3]
   addps    xmm6, xmm7            ; XMM6=[R32*v2+R33*v3,R22*v2+R23*v3,
                                  ;  R12*v2+R13*v3,R02*v2+R03*v3]
   addps    xmm4, xmm6            ; XMM4=New rotated vertex
   movntps  [eax+8*ecx], xmm4     ; Store rotated vertex to rotv.
   add      ecx, 2                ; Decrement the # of QWORDs to rotate by 2.
   jnz      rotate_vertices_loop
;============================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE
;  WAS ENTERED
; REGISTERS EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   pop edi
```

```
   pop esi
   pop ebx
   mov esp, ebp
   pop ebp
;============================================================================
   ret
_matrix_x_vector_sse ENDP
_TEXT   ENDS
END
```

To greatly enhance performance, the previous function can perform the matrix multiplication not only upon one four-column vector, but upon many. Creating a separate function to transform a single vertex and repeatedly calling the function is prohibitively expensive because of the overhead in pushing and popping registers from the stack. This applies to routines that negate a single vector, nullify a single vector, and add two vectors. Listing 28 is the 3DNow! technology counterpart to Listing 27 on page 229.

Listing 28. 4 × 4 Matrix Multiplication (3DNow!™ Technology)

```
; matrix_x_vector_3dnow(float *trR, float *v, int num_vertices_to_rotate,
   float *rotv);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c matrix_x_vector_3dnow.asm
;
.586
.K3D
.XMM
_TEXT   SEGMENT
PUBLIC _matrix_x_vector_3dnow
_matrix_x_vector_3dnow PROC NEAR
;============================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
;  ENTERED.
; REGISTERS EAX, ECX, AND EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED,
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   push ebp
   mov  ebp, esp
;============================================================================
; Parameters passed into routine:
;  [ebp+8]  = ->trR
;  [ebp+12] = ->v
;  [ebp+16] = num_vertices_to_rotate
;  [ebp+20] = ->rotv
;============================================================================
   push ebx
   push esi
   push edi
;============================================================================
; THE 4 ASM LINES BELOW LOAD THE FUNCTION'S ARGUMENTS INTO GENERAL-PURPOSE
;  REGISTERS (GPRs)
;  eax = address of Transposed Rotation Matrix
```

```
;  edx = address of vertices to rotate
;  ecx = # of vertices to rotate
;  ebx = address of rotated vertices
;===========================================================================
   mov eax, [ebp+8]    ; ESI = ->R
   mov edx, [ebp+12]   ; EDI = ->v
   mov ecx, [ebp+16]   ; ECX = num_vertices_to_rotate
   mov ebx, [ebp+20]   ; EAX = ->rotv

   femms               ; Clear MMX state.
ALIGN 16               ; Ensure optimal branch alignment.
;===========================================================================
; THIS LOOP ROTATES "num_vertices_to_rotate" VERTICES BY THE TRANSPOSED
;  ROTATION MATRIX "R" PASSED INTO THE ROUTINE AND STORES THE ROTATED
;  VERTICES TO "rotv".
;===========================================================================
rotate_vertices_loop:
   add         ebx,16         ; Increment ->v to next vertex.
   movq        mm0,[edx]      ; MM0 = [y,x]
   movq        mm1,[edx+8]    ; MM1 = [w,z]
   add         edx,16         ; Increment ->rotv to next transformed vertex.
   movq        mm2,mm0        ; MM2 = [y,x]
   movq        mm3,[eax]      ; MM3 = [R01,R00]
   punpckldq   mm0,mm0        ; MM0 = [x,x]
   movq        mm4,[eax+16]   ; MM4 = [R11,R10]
   pfmul       mm3,mm0        ; MM3 = [x*R01,x*R00]
   punpckhdq   mm2,mm2        ; MM2 = [y,y]
   pfmul       mm4,mm2        ; MM4 = [y*R11,y*R10]
   movq        mm5,[eax+8]    ; MM5 = [R03,R02]
   movq        mm7,[eax+24]   ; MM7 = [R13,R12]
   movq        mm6,mm1        ; MM6 = [w,z]
   pfmul       mm5,mm0        ; MM5 = [x*R03,x*R02]
   movq        mm0,[eax+32]   ; MM0 = [R21,R20]
   punpckldq   mm1,mm1        ; MM1 = [z,z]
   pfmul       mm7,mm2        ; MM7 = [y*R13,y*R12]
   movq        mm2,[eax+40]   ; MM2 = [R23,R22]
   pfmul       mm0,mm1        ; MM0 = [z*R21,z*R20]
   pfadd       mm3,mm4        ; MM3 = [x*R01+y*R11,x*R00+y*R10]
   movq        mm4,[eax+48]   ; MM4 = [R31,R30]
   pfmul       mm2,mm1        ; MM2 = [z*R23,z*R22]
   pfadd       mm5,mm7        ; MM5 = [x*R03+y*R13],x*R02+y*R12]
   movq        mm1,[eax+56]   ; MM1 = [R33,R32]
   punpckhdq   mm6,mm6        ; MM6 = [w,w]
   pfadd       mm3,mm0        ; MM3 = [x*R01+y*R11+z*R21,x*R00+y*R10+z*R20]
   pfmul       mm4,mm6        ; MM4 = [w*R31,w*R30]
   pfmul       mm1,mm6        ; MM1 = [w*R33,w*R32]
   pfadd       mm5,mm2        ; MM5 = [x*R03+y*R13+z*R23,x*R02+y*R12+z*R22]
   pfadd       mm3,mm4        ; MM3 = [x*R01+y*R11+z*R21+w*R31,
                              ;  x*R00+y*R10+z*R20+w*R30]
   movntq      [ebx-16],mm3   ; Store lower quadword of transformed vertex.
   pfadd       mm5,mm1        ; MM3 = [x*R03+y*R13+z*R23+w*R33,
```

```
                              ;  x*R02+y*R12+z*R22+w*R32]
   movntq      [ebx-8],mm5    ; Store upper QWORD of transformed vertex.
   dec         ecx            ; Decrement # of vertices to transform.
   jnz         rotate_vertices_loop
   femms                      ; Clear MMX state.
;===========================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE
;  WAS ENTERED.
; REGISTERS EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
;  WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   pop edi
   pop esi
   pop ebx
   mov esp, ebp
   pop ebp
;===========================================================================
   ret
_matrix_x_vector_3dnow ENDP
_TEXT   ENDS
END
```

Chapter 10 x87 Floating-Point Optimizations

AMD Athlon™ 64 and AMD Opteron™ processors support multiple methods of performing floating-point operations. They support the older x87 assembly instructions in addition to the more recent SIMD instructions (SSE, SSE2, and 3DNow!™ technologies). Many of the suggestions in this chapter are also generally applicable to the AMD Athlon 64 and AMD Opteron processors, with the exception of SSE2 optimizations and expanded register usage.

AMD Athlon 64 and AMD Opteron processors are 64-bit processors that are fully backwards compatible with 32-bit code. In general, 64-bit operating systems support the x87 and 3DNow! instructions in 32-bit threads; however, 64-bit operating systems may not support x87 and 3DNow! instructions in 64-bit threads. To make it easier to later migrate from 32-bit to 64-bit code, you may want to avoid x87 and 3DNow! instructions altogether and use only SSE and SSE2 instructions when writing new 32-bit code.

This chapter details the methods used to optimize floating-point code to the pipelined x87 floating-point registers.

This chapter covers the following topics:

10.1 Using Multiplication Rather Than Division

Optimization

If accuracy requirements allow, convert floating-point division by a constant to multiplication by the reciprocal.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Divisors that are powers of two—and their reciprocals—are exactly representable, and therefore do not cause an accuracy issue, except for the rare case in which the reciprocal overflows or underflows. Unless such an overflow or underflow occurs, always convert a division by a power of two for multiplication. Although the AMD Athlon 64 and AMD Opteron processors have high-performance division, multiplication is significantly faster than division.

10.2 Achieving Two Floating-Point Operations per Clock Cycle

Optimization

Pay special attention to the order and packing of the operations to sustain up to two floating-point operations per clock cycle.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The floating-point unit in the AMD Athlon, AMD Athlon 64, and AMD Opteron processors can sustain up to two floating-point operations per clock cycle. However, to achieve this, you must pay special attention to the order and packing of the operations. For example, consider multiplying a 30×30 double-precision matrix **A** by a transposed 30×30 double-precision matrix **B**, the result of which is called **C**.

Use Efficient Addressing of FPU Data When Loading and Storing

The rows of **A** are 240 bytes wide, as are the columns of **B**. There are eight x87 floating-point registers [ST(0)–ST(7)], and in this example, six rows of **A** are concurrently multiplied by a single column of **B**. The address of the first element of the first row of **A** (A[0]) is presumed to be stored in the EDI register, while the address of the first element of the first column of **B** (B[0]) is stored in ESI.

This addressing scheme might seem like a good idea, but it's not. Only 128 bytes can be addressed forward of A[0] with 8-bit offsets, meaning the size of the instructions are only 3 bytes (2 bytes for the instruction and 1 byte for the offset to the address stored in the general-purpose register). Upon offsetting more than 128 bytes from the address in the general-purpose register, the size of the instruction increases from 3 bytes to 6 bytes (offsets larger than 128 bytes are represented by 32 bits rather than 8 bits). Large instruction sizes reduce the number of decoded operations to be executed within the pipes of the floating-point unit, and as such prevent us from achieving two floating-point operations per clock cycle. To alleviate this, the general-purpose registers EDI and ESI are offset by 128 bytes such that they contain the addresses of A[15] and B[15]. This is beneficial because data within 128 bytes (16 double-precision numbers) before or after these two locations can now be accessed with instructions that are 2–3 bytes in size. The next five rows of **A** can be efficiently addressed in terms of the first row. Storing the size of a single row of **A** (240 bytes) in the EAX

register, the size of three rows (720 bytes) in EBX, and the size of five rows (1200 bytes) in ECX, the first element of rows 0–5 of **A** can be addressed as follows:

```
fld QWORD PTR [edi-128]         ; Load A[i,0].
fld QWORD PTR [edi+eax-128]     ; Load A[i+1,0].
fld QWORD PTR [edi+eax*2-128]   ; Load A[i+2,0].
fld QWORD PTR [edi+ebx-128]     ; Load A[i+3,0].
fld QWORD PTR [edi+eax*4-128]   ; Load A[i+4,0].
fld QWORD PTR [edi+ecx-128]     ; Load A[i+5,0].
```

This addressing scheme reduces the size of all loads from memory to 3 bytes; additionally, to address rows 6–11 of **A**, you only need to add 240*6 to EDI.

Avoid Register Dependencies by Spacing Apart Instructions that Accumulate Results in a Register

The second general optimization to consider is spacing out register dependencies. Operations internally in the floating-point unit have an execution latency (normally 3–4 cycles for x87 operations). Consider this instruction sequence:

```
fldz                        ; Push 0.0 onto floating-point stack.
fld  QWORD PTR [edi-128]    ; Push A[i,0] onto stack.
fmul QWORD PTR [esi-128]    ; Multiply A[i,0] by B[0,j].
faddp st(1), st(0)          ; Accumulate contribution to dot product of
                            ;  A's row i and B's column j.
fld  QWORD PTR [edi-120]    ; Push A[i,1] onto stack.
fmul QWORD PTR [esi-120]    ; Multiply A[i,1] by B[1,j].
faddp st(1), st(0)          ; Accumulate contribution to dot product of
                            ;  A's row i and B's column j.
fld  QWORD PTR [edi-112]    ; Push A[i,2] onto stack.
fmul QWORD PTR [esi-112]    ; Multiply A[i,2] by B[2,j].
faddp st(1), st(0)          ; Accumulate contribution to dot product of
                            ;  A's row i and B's column j.
```

The second statement loads A[0] into ST(0), and the third statement multiplies it by B[0]. The subsequent line adds this product to ST(1), where the dot product of row 0 of matrix **A** and column 0 of matrix **B** is accumulated. Each of the subsequent groups of three instructions adds the contribution of the remaining 29 elements to the dot product. This code is poor because all the addition operations depend upon the contents of a single register, ST(1). The AMD Athlon, AMD Athlon 64 and AMD Opteron processors have out-of-order-execution floating-point units, but none of the addition operations can be performed out of order because the result of each addition operation depends on the outcome of the previous addition operation. Instruction scheduling based on this code greatly limits the throughput of the floating-point unit. To alleviate this, space out operations that are dependent on one another. In this case, work with six rows of **A** rather than one at a time, as follows:

```
; Multiply first element of each of six rows of A by first element of
;  B's column j.
fldz                        ; Push 0.0 six times onto floating-point stack.
fldz
```

```
fldz
fldz
fldz
fldz
fld  QWORD PTR [esi-128]   ; Push B[0,j] onto stack.

fld  QWORD PTR [edi-128]          ; Push A[i,0] onto stack.
fmul st(0), st(1)                 ; Multiply A[i,0] by B[0,j].
faddp st(7), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i and B's column j.

fld   QWORD PTR [edi+eax-128]     ; Push A[i+1,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+1,0] by B[0,j].
faddp st(6), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+1 and B's column j.

fld   QWORD PTR [edi+eax*2-128]   ; Push A[i+2,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+2,0] by B[0,j].
faddp st(5), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+2 and B's column j.

fld   QWORD PTR [edi+ebx-128]     ; Push A[i+3,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+3,0] by B[0,j].
faddp st(4), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+3 and B's column j.

fld   QWORD PTR [edi+eax*4-128]   ; Push A[i+4,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+4,0] by B[0,j].
faddp st(3), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+4 and B's column j.

fmul  QWORD PTR [edi+ecx-128]     ; Multiply A[i+5,0] by B[0,j].
faddp st(1), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+5 and B's column j.
```

The processor can execute the instructions in this code sequence out of order because the instructions are independent. Even though the loads and multiplies are performed sequentially, the floating-point scheduler can execute the FLD and FMUL instructions out of order in addition to the FADD instruction so as to keep the multiplier and adder pipes of the floating-point unit busy. B[0] is initially loaded into an x87 register and multiplied by the loaded elements of each row with the *reg, reg* form of FMUL to minimize the number of load operations that need to be performed. Additionally, the first element from the sixth row of **A** is not loaded but simply multiplied from memory by the loaded element of B[0]. This eliminates an FLD instruction and decreases the number of instructions in the instruction cache and the workload on the processor's decoder. To achieve two floating-point operations per clock cycle, the number of floating-point operations should be twice the number of load-store operations. In the example above, there are 12 floating-point operations and seven operations requiring loads from memory, so nearly two floating-point operations can be performed per clock cycle.

Align and Pack DirectPath x87 Instructions

The last optimization to be performed is code packing and alignment. Having an abundance of operations in the decoder keeps the processor's schedulers well fed in circumstances where instructions cannot be immediately provided to the decoders. Floating-point x87 code can be aligned to 8-byte boundaries as illustrated here, which is optimal on AMD Athlon, AMD Athlon 64, and AMD Opteron processors:

```
;Instruction Address        Opcode      Instruction
;=============================================
 00000360                   66          DB      066h
 00000361                   DD 06       fld     QWORD PTR [esi]
 00000363                   66          DB      066h
 00000364                   DD 07       fld     QWORD PTR [edi]
 00000366                   D8 C9       fmul    st(0), st(1)

 00000368                   DE C7       faddp   st(7), st(0)
 0000036A                   DD 04 38    fld     QWORD PTR [edi+eax]
 0000036D                   66          DB      066h
 0000036E                   D8 C9       fmul    st(0), st(1)

 00000370                   DE C6       faddp   st(6), st(0)
 00000372                   DD 04 47    fld     QWORD PTR [edi+eax*2]
 00000375                   66          DB      066h
 00000376                   D8 C9       fmul    st(0), st(1)

 00000378                   DE C5       faddp   st(5), st(0)
 0000037A                   DD 04 3B    fld     QWORD PTR [edi+ebx]
 0000037D                   66          DB      066h
 0000037E                   D8 C9       fmul    st(0), st(1)

 00000380                   DE C4       faddp   st(4), st(0)
 00000382                   DD 04 87    fld     QWORD PTR [edi+eax*4]
 00000385                   66          DB      066h
 00000386                   D8 C9       fmul    st(0), st(1)

 00000388                   DE C3       faddp   st(3), st(0)
 0000038A                   DC 0C 39    fmul    QWORD PTR [edi+ecx]
 0000038D                   66          DB      066h
 0000038E                   DE C1       faddp   st(1), st(0)
```

The instruction address specifies the address (in hexadecimal) of the instruction to the right.

Typically three DirectPath instructions occupy 7 bytes. Maintaining 8-byte alignment for the next group of three instructions requires the addition of a single byte. A 1-byte padding can easily be achieved using the single-byte NOP instruction (opcode 90h), as recommended in "Code Padding with Operand-Size Override and NOP" on page 85. However, for the special case of x87 instructions,

the operand-size override (66h) serves as a high-performance NOP instruction and is the recommended choice for padding an x87 instruction without altering its behavior, as shown here:

```
DB 066h   ; Operand-size override used as high-performance NOP instruction
```

This usage of the operand-size override alone as a filler byte (without an accompanying NOP instruction) is permitted only for x87 instructions. This usage of the operand-size override can be applied to all but four of the x87 instructions. The FLDENV, FRSTOR, FSTENV, and FSAVE instructions and their no-wait forms behave differently when associated with an operand-size override; therefore, these should not be padded with the operand-size override.

10.3 Floating-Point Compare Instructions

Optimization

For branches that are dependent on floating-point comparisons, use the FCOMI, FCOMIP, FUCOMI, and FUCOMIP instructions:

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The FCOMI, FCOMIP, FUCOMI, and FUCOMIP instructions are much faster than the classical approach using FSTSW. When FSTSW cannot be avoided (for example, backward compatibility of code with older processors), no floating-point instruction should occur between an FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FUCOM, FUCOMP, FUCOMPP, or FTST instruction and a dependent FSTSW instruction. This optimization allows the use of a fast-forwarding mechanism for the floating-point condition codes internal to the processor's floating-point unit and increases performance.

10.4 Using the FXCH Instruction Rather Than FST/FLD Pairs

Optimization

Increase parallelism by breaking up dependency chains or by evaluating multiple dependency chains simultaneously by explicitly switching execution between them.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Although the AMD Athlon 64 and AMD Opteron processor's floating-point unit 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 four-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 micro-op. The floating-point unit of the AMD Athlon 64 and AMD Opteron processors 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 macro-ops.

10.5 Floating-Point Subexpression Elimination

Optimization

Reduce the number of superfluous FXCH instructions by putting the shared source operand at the top of the stack to eliminate subexpressions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

There are cases that 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:

Examples

Listing 29. Avoid

```
;=====================
; func((x*y),(x+z))
;=====================
fld   x           ; x
fld   y           ; y x
fld   x           ; x y x
fld   z           ; z x y x
faddp st(1), st   ; x+z y x
fxch  st(2)       ; x y x+z
fmulp st(1), st   ; x*y x+z
```

Listing 30. Preferred

```
fld   z           ; z
fld   y           ; y z
fld   x           ; x y z
fmul st(1), st   ; x x*y z
faddp st(2), st   ; x*y x+z
```

10.6 Accumulating Precision-Sensitive Quantities in x87 Registers

Optimization

Accumulate results in the x87 registers rather than the SSE and SSE2 XMM registers, if more than 64 bits of accuracy are required.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

More than 64 bits of accuracy may be required, as when accumulating a result (for example, during the calculation of dot product). The precision of floating-point operations in the x87 registers ST(0)–ST(7) is 80 bits internally, whereas the precision of operations using SIMD instructions is only 64 bits.

10.7 Avoiding Extended-Precision Data

Optimization

Store floating-point data in single-precision or double-precision format.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Loading and storing extended-precision data is significantly slower than storing single- or double-precision data.

Appendix A Microarchitecture for AMD Athlon™ 64 and AMD Opteron™ Processors

When discussing processor design, it is important to understand the terms *architecture*, *microarchitecture*, and *design implementation*.

The *architecture* consists of the instruction set and those features of a processor that are visible to software programs running on the processor. The architecture determines what software the processor can run. The AMD64 architecture of the AMD Athlon™ 64 and AMD Opteron™ processors is compatible with the industry-standard x86 instruction set.

The term *microarchitecture* refers to the design features used to reach the target cost, performance, and functionality goals of the processor. The AMD64 architecture employs a decoupled decode/execution design approach. In other words, decoders and execution units essentially operate independently; the execution core uses a small number of instructions and simplified circuit design for fast single-cycle execution and fast operating frequencies.

The *design implementation* refers to a particular combination of physical logic and circuit elements that comprise a processor that meets the microarchitecture specifications.

This appendix covers the following topics:

A.1 Key Microarchitecture Features

The AMD Athlon 64 and AMD Opteron processors include many features designed to improve software performance. The internal design, or *microarchitecture*, of these processors provides the following key features:

- Integrated DDR memory controller
- 64-Kbyte L1 instruction cache and 64-Kbyte L1 data cache
- On-chip L2 cache
- Instruction predecode and branch prediction during cache-line fills
- Decoupled decode/execution core
- Three-way AMD64 instruction decoding
- Dynamic scheduling and speculative execution
- Three-way integer execution
- Three-way address generation
- Three-way floating-point execution
- 3DNow!™ technology, MMX™, SSE, and SSE2 single-instruction multiple-data (SIMD) instruction extensions
- Superforwarding
- Deep out-of-order integer and floating-point execution
- In 64-bit mode, eight additional XMM registers (for use with SSE and SSE2 instructions) and eight additional general-purpose registers (GPRs)
- HyperTransport™ technology

A.2 Microarchitecture forAMD Athlon™ 64 and AMD Opteron™ Processors

The AMD Athlon 64 and AMD Opteron processors implement the AMD64 instruction set by means of *micro-ops*—simple fixed-length operations designed to include direct support for AMD64 instructions and adhere to the high-performance principles of fixed-length encoding, regularized instruction fields, and a large register set. The enhanced microarchitecture enables higher processor core performance and promotes straightforward extensibility for future designs.

A.3 Superscalar Processor

The AMD Athlon 64 and AMD Opteron processors are aggressive, out-of-order, three-way superscalar AMD64 processors. They can fetch, decode, and issue up to three AMD64 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 micro-ops to the three general-purpose integer execution units (ALUs), three address-generation units (AGUs), and three floating-point execution units. The processors move integer instructions down the integer execution pipeline, which consists of the integer scheduler and the ALUs, as shown in Figure 3 on page 250. Floating-point instructions are handled by the floating-point execution pipeline, which consists of the floating-point scheduler and the floating-point execution units.

A.4 Processor Block Diagram

A block diagram of the AMD Athlon 64 and AMD Opteron processors is shown in Figure 3 on page 250.

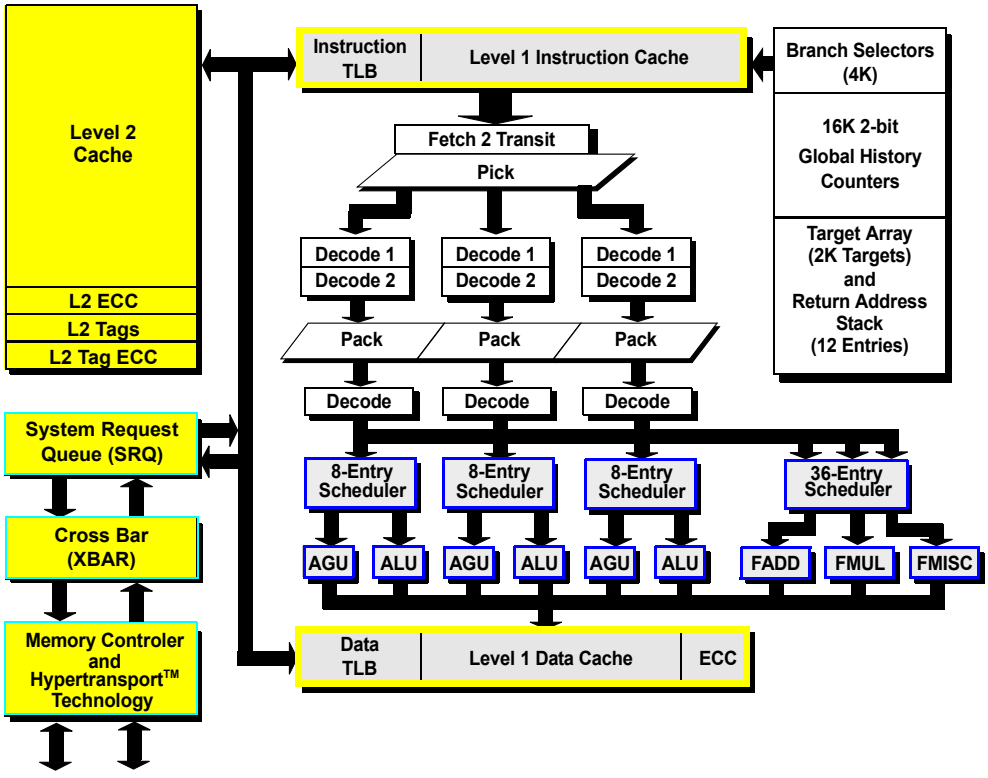


Figure 3. AMD Athlon™ 64 and AMD Opteron™ Processors Block Diagram

A.5 L1 Instruction Cache

The out-of-order execution engine of the AMD Athlon 64 and AMD Opteron processors contains a very large L1 instruction cache. Each line in this cache 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 L2 cache or, subsequently, from the local memory using the integrated memory controller.

The L1 instruction cache generates fetches on the naturally aligned 64 bytes containing the instructions and the next sequential line of 64 bytes (a prefetch). The principle of program-spatial locality makes code prefetching very effective and avoids or reduces execution stalls caused by the amount of time required to read the necessary code. Cache-line replacement is based on a least-recently-used replacement algorithm.

Table 8 provides specifications on the L1 instruction cache for various AMD processors.

Table 8. L1 Instruction Cache Specifications by Processor

Processor name	Family	Model	Associativity	Size (Kbytes)
AMD Athlon™ XP processor	6	6	2 ways	64
AMD Athlon™ 64 processor	15	0	2 ways	64
AMD Opteron™ processor	15	1	2 ways	64

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 AMD64 instructions.

A.6 Branch-Prediction Table

The fetch logic accesses the branch prediction table in parallel with the L1 instruction cache. The information stored in the branch prediction table is used to predict the direction of branch instructions. When instruction cache lines are evicted to the L2 cache, branch selectors and predecode information are also stored in the L2 cache.

The AMD Athlon 64 and AMD Opteron processors employ combinations of a branch target address buffer (BTB), a global history bimodal counter (GHBC) table, and a return address stack (RAS) 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 misprediction, the minimum penalty is 10 cycles.

The BTB is a 2048-entry table that caches in each entry the predicted target address of a branch. The 16384-entry GHBC table contains 2-bit saturating counters used to predict whether a conditional branch is taken. The GHBC table is indexed using the outcome (taken or not taken) of the last eight conditional branches and 4 bits of the branch address. The GHBC table allows the processors to predict branch patterns of up to eight branches.

In addition, the processors implement a 12-entry return address stack to predict return addresses from a near or far call. As calls are fetched, the next rIP is pushed onto the return stack. Subsequent returns pop a predicted return address off the top of the stack.

A.7 Fetch-Decode Unit

The fetch-decode unit performs early decoding of AMD64 instructions into macro-ops. The outputs of the early decoders keep all (DirectPath or VectorPath) instructions in program order. Early decoding produces three macro-ops 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. Decoding a VectorPath instruction may prevent simultaneously decoding of a DirectPath instruction.

When the target 16-byte instruction window is obtained from the L1 instruction cache, the instruction bytes are examined to determine whether the type of basic decode to occur is DirectPath or VectorPath.

A.8 Instruction Control Unit

The *instruction control unit* (ICU) is the control center for the AMD Athlon 64 and AMD Opteron processors. It controls the centralized in-flight reorder buffer, the integer scheduler, and the floating-point scheduler. In turn, the ICU is responsible for the following functions: macro-op dispatch, macro-op retirement, register and flag dependency resolution and renaming, execution resource management, interrupts, exceptions, and branch mispredictions.

The instruction control unit takes the three macro-ops 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 macro-ops each. The reorder buffer allows the instruction control unit to track and monitor up to 72 in-flight macro-ops (whether integer or floating-point) for maximum instruction throughput. The instruction control unit can simultaneously dispatch multiple macro-ops from the reorder buffer to both the integer and floating-point schedulers for final decode, issue, and execution as micro-ops. In addition, the instruction control unit handles exceptions and manages the retirement of macro-ops.

A.9 Translation-Lookaside Buffer

A *translation-lookaside buffer* (TLB) is a special on-chip cache that holds a table that matches the most-recently-used virtual addresses to their physical addresses.

The AMD Athlon 64 and AMD Opteron processors utilize a two-level TLB structure. A flush filter—new on the AMD Athlon 64 and AMD Opteron processors—eliminates unnecessary TLB flushes when loading the CR3 register.

L1 Instruction TLB Specifications

Table provides the specifications of the L1 instruction TLB for various AMD processors.

Table 9. L1 Instruction TLB Specifications

				Number of Entries	
Processor Name	**Family**	**Model**	**Associativity**	**2-Mbyte Pages[1]**	**4-Kbyte Pages**
AMD Athlon™ XP Processor	6	6	Full	8	16

Note:

1. *The number of entries available for 4-Mbyte pages is one-half this value (4-Mbyte pages require two 2-Mbyte entries).*

Table 9. L1 Instruction TLB Specifications

				Number of Entries	
Processor Name	**Family**	**Model**	**Associativity**	**2-Mbyte Pages[1]**	**4-Kbyte Pages**
AMD Athlon™ 64 Processor	15	4	Full	8	32
AMD Opteron™ Processor	15	5	Full	8	32

Note:

1. *The number of entries available for 4-Mbyte pages is one-half this value (4-Mbyte pages require two 2-Mbyte entries).*

L1 Data TLB Specifications

Table 10 provides the specifications of the L1 data TLB for various AMD processors.

Table 10. L1 Data TLB Spcifications

				Number of Entries	
Processor Name	**Family**	**Model**	**Associativity**	**2-Mbyte pages[1]**	**4-Kbyte pages**
AMD Athlon™ XP Processor	6	6	Full	8	32
AMD Athlon™ 64 Processor	15	4	Full	8	32
AMD Opteron™ Processor	15	5	Full	8	32

Note:

1. *The number of entries available for 4-Mbyte pages is one-half this value (4-Mbyte pages require two 2-Mbyte entries).*

L2 TLB Specifications

Table 11 provides the specifications on the L2 TLB for various AMD processors.

Table 11. L2 TLB Specifications by Processor

Processor Name	**Family**	**Model**	**Associativity**	**Number of Entries (4-Kbyte Pages)**
AMD Athlon™ XP Processor	6	6	4 ways	256
AMD Athlon™ 64 Processor	15	4	4 ways	512
AMD Opteron™ Processor	15	5	4 ways	512

A.10 L1 Data Cache

The L1 data cache contains two 64-bit ports. It is a write-allocate and writeback cache that uses a least-recently-used replacement policy. It is divided into eight banks, each eight bytes wide. In addition, the L1 cache supports the MOESI (Modified, Owner, Exclusive, Shared, and Invalid) cache-coherency protocol and data parity.

Table 12 provides specifications on the L1 data cache for various AMD processors.

Table 12. L1 Data Cache Specifications by Processor

Processor name	Family	Model	Associativity	Size (Kbytes)
AMD Athlon™ XP Processor	6	6	2 ways	64
AMD Athlon™ 64 Processor	15	4	2 ways	64
AMD Opteron™ Processor	15	5	2 ways	64

A.11 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. The reservation stations are eight entries deep, for a total queuing system of 24 integer macro-ops. Each reservation station divides the macro-ops into integer and address generation micro-ops, as required.

A.12 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—or arithmetic-logic unit (ALU)—and an address generation unit (AGU). The integer execution pipeline is organized to match the three macro-op dispatch pipes in the ICU as shown in Figure 4.

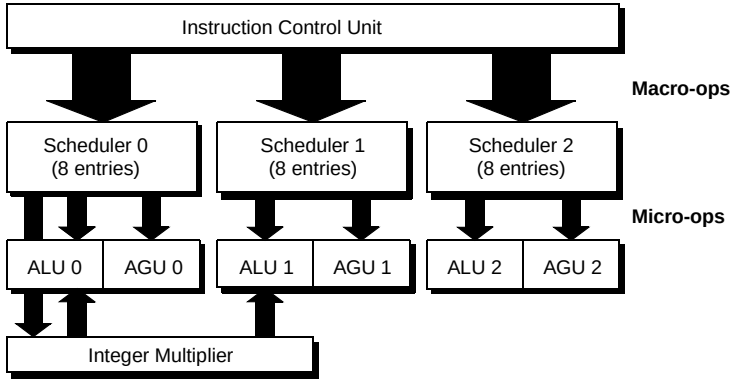


Figure 4. Integer Execution Pipeline

Macro-ops are broken down into micro-ops in the schedulers. Micro-ops issue when their operands are available either from the register file or result buses. Micro-ops are executed when their operands are available. Micro-ops from a single operation can execute out-of-order. In addition, a particular

integer pipe can execute two micro-ops from different macro-ops (one in the ALU and one in the AGU) at the same time. See Figure 5 on page 255.

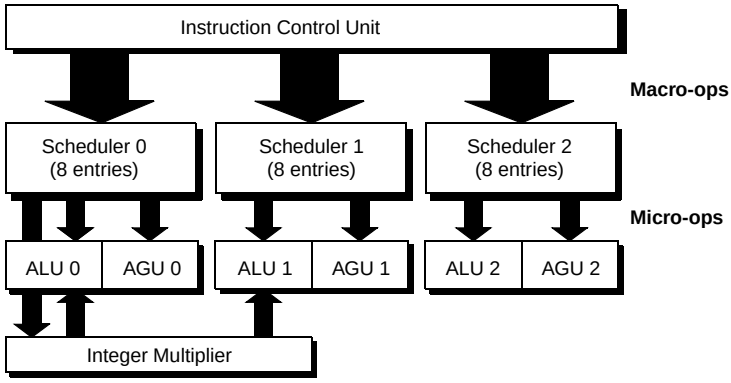


Figure 5. Integer Execution Unit

Each of the three ALUs performs general purpose 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 micro-ops for a given macro-op are executed.

All integer operations can be handled within any of the three ALUs with the exception of multiplies. Multiplies are handled by a pipelined multiplier that is attached to the pipeline at pipe 0, as shown in Figure 5. Multiplies always issue to integer pipe 0, and the issue logic creates results bus bubbles for the multiplier in integer pipes 0 and 1 by preventing non-multiply micro-ops from issuing at the appropriate time.

A.13 Floating-Point Scheduler

The floating-point logic of the AMD Athlon 64 and AMD Opteron processors is a high-performance, fully pipelined, superscalar, out-of-order execution unit. It is capable of accepting three macro-ops per cycle from any mixture of the following types of instructions:

- x87 floating-point
- 3DNow! technology
- MMX technology
- SSE
- SSE2

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

Superforwarding is a performance optimization. It allows a floating point operation having a dependency on a register to be scheduled sooner when that register is waiting to be filled by a pure load from memory. Instead of waiting for the first instruction to write its load-data to the register and then waiting for the second instruction to read it, the load-data can be provided directly to the dependent instruction, much like regular forwarding between FPU-only operations. The result from the load is said to be "superforwarded" to the floating-point operation. In the following example, the FADD can be scheduled to execute as soon as the load operation fetches its data rather than having to wait and read it out of the register file.

```
fld    [somefloat]       ;Load a floating point
                          ;value from memory into ST(0)
fadd   st(0),st(1)       ;The data from the load will be
                          ;forwarded directly to this instruction,
                          ;no need to read the register file
```

A.14 Floating-Point Execution Unit

The floating-point execution unit (FPU) is implemented as a coprocessor having its own out-of-order control in addition to the data path. The FPU handles all register operations for x87 instructions, all 3DNow! technology operations, all MMX operations, and all SSE and SSE2 operations. The FPU consists of a stack renaming unit, a register renaming unit, a scheduler, a register file, and three parallel execution units. Figure 6 shows a block diagram of the dataflow through the FPU.

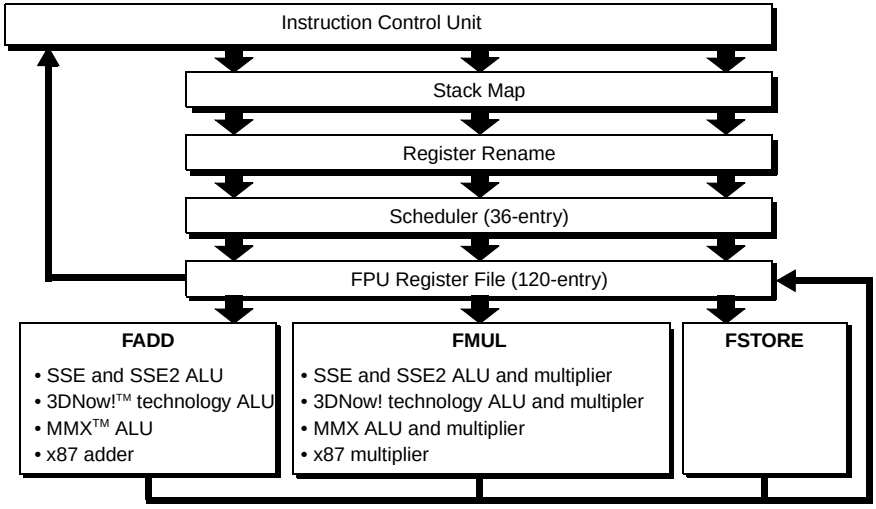


Figure 6. Floating-Point Unit

As shown in Figure 6, the floating-point logic uses three separate execution positions or pipes. The first of the three pipes is generally known as the adder pipe (FADD), and it contains an MMX ALU/shifter and floating-point add execution units. The second pipe is known as the multiplier (FMUL). It contains the floating-point multiplier/divider/square root unit and also an MMX ALU. The third pipe is known as the floating-point load/store (FSTORE), which handles floating-point stores and many micro-op primitives used in VectorPath sequences.

A.15 Load-Store Unit

The load-store unit (LSU) is shown in Figure 7. It manages data load and store accesses to the L1 data cache and, if required, to the 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 operations (and mix of loads and stores) per cycle. Up to two 64-bit stores can be performed per cycle. The 32-entry queue effectively holds requests that missed in the L1 cache probe by the 12-entry queue. Finally, the LSU helps ensure that the architectural load and store ordering rules are preserved (a requirement for AMD64 architecture compatibility).

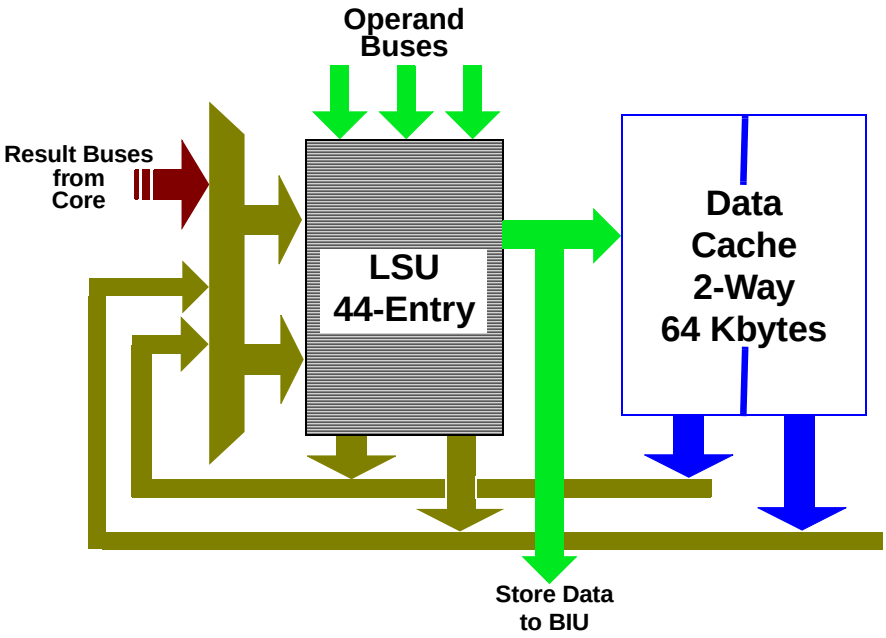


Figure 7. Load-Store Unit

A.16 L2 Cache

The AMD Athlon 64 and AMD Opteron processors each contain an integrated L2 cache. This full-speed on-die L2 cache features an exclusive cache architecture. The L2 cache contains only victim or copy-back cache blocks that are to be written back to the memory subsystem as a result of a conflict miss. These terms, victim or copy-back, refer to cache blocks that were previously held in the L1 cache but had to be overwritten (evicted) to make room for newer data. The victim buffer contains data evicted from the L1 cache.

The L2 cache in the AMD Athlon XP, AMD Athlon™ 64, and AMD Opteron processors is 16-way associative.

A.17 Write-combining

See Appendix B, “Implementation of Write-Combining,” on page 261 for detailed information about write-combining.

A.18 Buses for AMD Athlon™ 64 and AMD Opteron™ Processor

AMD Athlon 64 and AMD Opteron processors feature an integrated memory controller and HyperTransport technology for interfacing to I/O devices. These integrated features, along with other logic, bring the Northbridge functionality onto the processor.

A.19 Integrated Memory Controller

AMD Athlon 64 and AMD Opteron processors provide an integrated low-latency, high-bandwidth DDR memory controller.

The memory controller supports:

- DRAM devices that are 4, 8, and 16 bits wide.
- Interleaving memory within DIMMs.
- ECC checking with double-bit detection and single-bit correction.

The memory controller may be configured for 32-byte or 64-byte burst lengths.

For specifications on a certain processor's memory controller, see the data sheet for that processor. For information on how to program the memory controller, see the *BIOS and Kernel Developer's Guide for AMD Athlon™ 64 and AMD Opteron™ Processors*, order# 26094.

A.20 HyperTransport™ Technology Interface

HyperTransport technology is a scalable, high-speed, low-latency, point-to-point, packetized link that:

- Enables chips to transfer data at rates up to 12.8 Gbytes/s (6.4 Gbytes/s in each direction simultaneously with a 32-bit link).
- Simplifies connectivity by replacing legacy buses and bridges.
- Reduces latencies and bottlenecks within systems.

When compared with traditional technologies, HyperTransport technology allows much faster data-transfer rates. A 16-bit HyperTransport I/O link, for example, provides a maximum aggregate transfer rate of 6.4 Gbytes/s—48 times the peak transfer rate of a 33-MHz PCI bus. For more information on HyperTransport technology, see the *HyperTransport I/O Link Specification*, available at www.hypertransport.org.

HyperTransport™ Technology

On AMD Athlon 64 and AMD Opteron processors, HyperTransport technology provides the link to I/O devices. Some processor models—for example, those designed for use in multiprocessing systems—also utilize HyperTransport technology to connect to other processors. Table 13 lists the HyperTransport specifications of different AMD Athlon™ 64 and AMD Opteron™ processors.

Table 13. HyperTransport™ Specifications by Processor

Processor Name	Family	Model	Number of HyperTransport™ Links	Number of Bits Per Link	Transfer Rate Per Link (in each direction)	Width of Memory Data Bus (includes ECC bits)
AMD Athlon™ 64 Processor	15	4	2[1]	8	1.6 Gbytes/s	72
AMD Opteron™ Processor	15	5	3	16	3.2 Gbytes/s	144

Note:

1. *These two links can also be configured as a single 16-bit link providing 3.2 Gbytes/s in each direction.*

Appendix B Implementation of Write-Combining

This appendix describes the memory write-combining feature implemented in the AMD Athlon™ 64 and AMD Opteron™ processors. Write-combining is the merging of multiple memory write cycles that target locations within the address range of a write buffer.

The AMD Athlon 64 and AMD Opteron processors support 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.

This appendix covers the following topics:

B.1 Write-Combining Definitions and Abbreviations

This appendix uses the following definitions and abbreviations:

- MTRR—Memory type and range register
- PAT—Page attribute table
- 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
- Doubleword—32 bits
- Quadword—64 bits or 2 doublewords
- Octaword—128 bits or 2 quadwords
- Cache Block—64 bytes or 4 octawords or 8 quadwords

B.2 Programming Details

The steps required for programming write-combining on the AMD Athlon 64 and AMD Opteron processors are as follows:

1. Verify the presence of an AMD Athlon™ 64 or AMD Opteron 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 64 and AMD Opteron processors, the instruction family code is Fh.
2. Verify the presence of the MTRRs and the PAT extensions. The presence of the MTRRs is indicated by bit 12 and the presence of the PAT extensions 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. Enable write-combiningwrite-combining. Write-combining is controlled by the MTRRs and PAT extensions. Write-combining should be enabled for the appropriate memory ranges. For more information on the MTRRs and the PAT extensions, see volume 2 of the *AMD64 Architecture Programmer's Manual*, order# 24593.

B.3 Write-combining Operations

In order to improve system performance, the AMD Athlon 64 and AMD Opteron processors aggressively combine 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 processor continues to combine writes to this buffer without writing the data to the system, as long as certain rules apply (see Table 14 on page 263 for more information). The data sizes can be bytes, words, doublewords, or quadwords.

- WC memory type writes can be combined in any order up to a full 64-byte 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 14 on page 263. When combining is interrupted, one or more bus commands are issued to the system for that write buffer, as described in "Sending Write-Buffer Data to the System" on page 264.

Table 14. 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, and 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.

Table 14. Write-Combining Completion Events (Continued)

Event	Comment
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.

B.4 Sending Write-Buffer Data to the System

The maximum write combined throughput is achieved when all quadwords or doublewords are valid and the AMD Athlon 64 and AMD Opteron processors can use one efficient 64-byte memory write instead of multiple 8-byte memory writes.

Appendix C Instruction Latencies

This appendix provides a complete listing of all AMD64 instructions, along with their encodings, decode types, and execution latencies. For more information on these instructions, see volumes 3, 4, and 5 of the *AMD64 Architecture Programmer's Manual* (order# 24594, 26568, and 26569).

Note: *Some prior AMD documents referred to one group of instructions as MMX™ technology* extensions*. Those instructions are still supported by the AMD Athlon™ 64 and AMD Opteron™ processors, but are documented with the SSE instructions in this guide. (The MMX™ technology* instructions *remain a separate group.)*

The instruction entries in this appendix are grouped into categories as indicated in the following table and are presented within each category in alphabetical order by mnemonic:

C.1 Understanding Instruction Entries

To use the information in this appendix effectively, you need to understand how the entry for an instruction is organized and how to interpret certain items.

Example: Instruction Entry

The entry for an instruction begins with its syntax. Subsequent columns provide additional information about the instruction.

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
ADD *mreg8*, *reg8*	00h		11-xxx-xxx	DirectPath	1	

Parts of the Instruction Entry

This table describes the columns that are common to each instruction entry in this appendix.

Column	Description
Syntax	Shows the syntax for the instruction—the permitted arrangement of its parts. Items in italics are placeholders for operands that you must provide. For information on how to interpret the placeholders, see “Interpreting Placeholders” on page 267
Encoding	Shows how the assembler translates the instruction into machine language. Subcolumns show the individual bytes of the encoding.
Decode type	Shows the method that the processor uses to decode the instruction—either DirectPath Single (DirectPath), DirectPath Double (Double), or VectorPath.
Latency	Shows the static execution latency for the instruction. For details on how to interpret the latency information, see “Interpreting Latencies” on page 268.
Throughput	This value indicates the maximum theoretical rate of execution of that instruction. For example, a value of 1/2 means that one such instruction executes every two clocks, or two such instructions in four clocks and so on. A value of 3/1 indicates that three such instructions can be executed every clock, but fewer than three such instructions would still take one clock.

The entries for floating-point, MMX, SSE, and SSE2, and 3DNow!™ instructions have an additional column [FPU Pipe(s)] that lists the possible floating-point unit (FPU) pipelines available for use by any particular DirectPath or Double decoded operation. For example, the floating point multiplier is represented by FMUL. Because VectorPath instructions cannot be executed concurrently with any other instruction, their pipeline usage is irrelevant, but the information is listed where it is available.

Interpreting Placeholders

The Syntax column for an instruction entry shows the mnemonic for the instruction followed by any operands. Items in italics are placeholders for operands that you must provide. A placeholder indicates the size and type of operand that is allowed.

This operand	Is a placeholder for
disp8	A byte (8-bit) displacement value
disp16/32	A word (16-bit) or doubleword (32-bit) displacement value
disp32/48	A doubleword (32-bit) or 48-bit displacement value
imm8	A byte (8-bit) immediate value
imm16	A word (16-bit) immediate value
imm32	A doubleword (32-bit) immediate value
mem8	A byte (8-bit) memory location
mem16/32/64	A memory location that contains a word, doubleword, or quadword
mem16/32&mem16/32	A memory location that contains a pair of words or doublewords
mem32/48	A doubleword (32-bit) or 48-bit memory location
mem48	A 48-bit memory location
mem64	A quadword (64-bit) memory location
mem128	A double quadword (128-bit) memory location
mem32real	A memory location that contains a single-precision (32-bit) floating-point value
mem64real	A memory location that contains a double-precision (64-bit) floating-point value
mem80real	A memory location that contains a double-extended-precision (80-bit) floating-point value
mmreg	An MMX™ register
mmreg1	An MMX register defined by bits 5, 4, and 3 of the ModRM byte
mmreg2	An MMX register defined by bits 2, 1, and 0 of the ModRM byte
mreg8	A byte general-purpose register defined by the r/m field (bits 2, 1, and 0) of the ModRM byte
mreg16/32/64	A word, doubleword, or quadword general-purpose register defined by the r/m field (bits 2, 1, and 0) of the ModRM byte
reg8	A byte general-purpose register defined by instruction byte(s) or the reg field (bits 5, 4, and 3) of the ModRM byte
reg16/32/64	A word, doubleword, or quadword general-purpose register defined by instruction byte(s) or the reg field (bits 5, 4, and 3) of the ModRM byte
sreg	A segment register (always 16 bits wide)
xmmreg	An XMM register
xmmreg1	An XMM register defined by bits 5, 4, and 3 of the ModRM byte
xmmreg2	An XMM register defined by bits 2, 1, and 0 of the ModRM byte

Interpreting Latencies

The Latency column for an instruction entry shows the static execution latency for the instruction. The static execution latency is the number of clock cycles it takes to execute the serially dependent sequence of micro-ops that comprise the instruction.

The latencies in this appendix are estimates and are subject to change. They assume that:

- The instruction is an L1-cache hit that has already been fetched and decoded, with the operations loaded into the scheduler.
- Memory operands are assumed to be in the L1 data cache.
- There is no contention for execution resources or load-store unit resources.

The following formats are used to indicate the static execution latency:

Latency format	Description	Example
x	The latency is the indicated value.	3
x–*y*	The latency is a value greater than or equal to *x* and less than or equal to *y*.	31–73
x/*y*/*z*	The latency differs according to the size of the operands. The values *x*, *y*, and *z* are the 16-, 32-, and 64-bit latencies, respectively.	26/42/74
x (*y*)	The latency depends on whether an error condition exists. When there is no error condition, *x* is the latency. When an error condition exists, *y* is the latency.	68 (108)
~	The latency is unavailable.	

C.2 Integer Instructions

Table 15. Integer Instructions

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
AAA	37h			VectorPath	5	
AAD (or directly coded D5 *ib*, where *ib* is a byte value other than 0Ah)	D5h	0Ah		VectorPath	5	
AAM (or directly coded D4 *ib*, where *ib* is a byte value other than 0Ah)	D4h	0Ah		VectorPath	15	
AAS	3Fh			VectorPath	5	
ADC *mreg8*, *reg8*	10h		11-xxx-xxx	DirectPath	1	
ADC *mem8*, *reg8*	10h		mm-xxx-xxx	DirectPath	4	
ADC *mreg16/32/64*, *reg16/32/64*	11h		11-xxx-xxx	DirectPath	1	
ADC *mem16/32/64*, *reg16/32/64*	11h		mm-xxx-xxx	DirectPath	4	
ADC *reg8*, *mreg8*	12h		11-xxx-xxx	DirectPath	1	
ADC *reg8*, *mem8*	12h		mm-xxx-xxx	DirectPath	4	
ADC *reg16/32/64*, *mreg16/32/64*	13h		11-xxx-xxx	DirectPath	1	
ADC *reg16/32/64*, *mem16/32/64*	13h		mm-xxx-xxx	DirectPath	4	
ADC AL, *imm8*	14h			DirectPath	1	
ADC AX, *imm16*	15h			DirectPath	1	
ADC EAX, *imm32*	15h			DirectPath	1	
ADC RAX, *imm32* (sign extended)	15h			DirectPath	1	
ADC *mreg8*, *imm8*	80h		11-010-xxx	DirectPath	1	
ADC *mem8*, *imm8*	80h		mm-010-xxx	DirectPath	4	
ADC *mreg16/32/64*, *imm16/32*	81h		11-010-xxx	DirectPath	1	
ADC *mem16/32/64*, *imm16/32*	81h		mm-010-xxx	DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see "32/64-Bit vs. 16-Bit Forms of the LEA Instruction" on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in "Latency of Repeated String Instructions" on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
ADC *mreg16/32/64*, *imm8* (sign extended)	83h		11-010-xxx	DirectPath	1	
ADC *mem16/32/64*, *imm8* (sign extended)	83h		mm-010-xxx	DirectPath	4	
ADD *mreg8*, *reg8*	00h		11-xxx-xxx	DirectPath	1	
ADD *mem8*, *reg8*	00h		mm-xxx-xxx	DirectPath	4	
ADD *mreg16/32/64*, *reg16/32/64*	01h		11-xxx-xxx	DirectPath	1	
ADD *mem16/32/64*, *reg16/32/64*	01h		mm-xxx-xxx	DirectPath	4	
ADD *reg8*, *mreg8*	02h		11-xxx-xxx	DirectPath	1	
ADD *reg8*, *mem8*	02h		mm-xxx-xxx	DirectPath	4	
ADD *reg16/32/64*, *mreg16/32/64*	03h		11-xxx-xxx	DirectPath	1	
ADD *reg16/32/64*, *mem16/32/64*	03h		mm-xxx-xxx	DirectPath	4	
ADD AL, *imm8*	04h			DirectPath	1	
ADD AX, *imm16*	05h			DirectPath	1	
ADD EAX, *imm32*	05h			DirectPath	1	
ADD RAX, *imm32* (sign extended)	05h			DirectPath	1	
ADD *mreg8*, *imm8*	80h		11-000-xxx	DirectPath	1	
ADD *mem8*, *imm8*	80h		mm-000-xxx	DirectPath	4	
ADD *mreg16/32/64*, *imm16/32*	81h		11-000-xxx	DirectPath	1	
ADD *mem16/32/64*, *imm16/32*	81h		mm-000-xxx	DirectPath	4	
ADD *mreg16/32/64*, *imm8* (sign extended)	83h		11-000-xxx	DirectPath	1	
ADD *mem16/32/64*, *imm8* (sign extended)	83h		mm-000-xxx	DirectPath	4	
AND *mreg8*, *reg8*	20h		11-xxx-xxx	DirectPath	1	
AND *mem8*, *reg8*	20h		mm-xxx-xxx	DirectPath	4	
AND *mreg16/32/64*, *reg16/32/64*	21h		11-xxx-xxx	DirectPath	1	
AND *mem16/32/64*, *reg16/32/64*	21h		mm-xxx-xxx	DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
AND *reg8*, *mreg8*	22h		11-xxx-xxx	DirectPath	1	
AND *reg8*, *mem8*	22h		mm-xxx-xxx	DirectPath	4	
AND *reg16/32/64*, *mreg16/32/64*	23h		11-xxx-xxx	DirectPath	1	
AND *reg16/32/64*, *mem16/32/64*	23h		mm-xxx-xxx	DirectPath	4	
AND AL, *imm8*	24h			DirectPath	1	
AND AX, *imm16*	25h			DirectPath	1	
AND EAX, *imm32*	25h			DirectPath	1	
AND RAX, *imm32* (sign extended)	25h			DirectPath	1	
AND *mreg8*, *imm8*	80h		11-100-xxx	DirectPath	1	
AND *mem8*, *imm8*	80h		mm-100-xxx	DirectPath	4	
AND *mreg16/32/64*, *imm16/32*	81h		11-100-xxx	DirectPath	1	
AND *mem16/32/64*, *imm16/32*	81h		mm-100-xxx	DirectPath	4	
AND *mreg16/32/64*, *imm8* (sign extended)	83h		11-100-xxx	DirectPath	1	
AND *mem16/32/64*, *imm8* (sign extended)	83h		mm-100-xxx	DirectPath	4	
ARPL *mreg16*, *reg16*	63h		11-xxx-xxx	VectorPath	13	
ARPL *mem16*, *reg16*	63h		mm-xxx-xxx	VectorPath	18	
BOUND *reg16/32*, *mem16/32&mem16/32*	62h		mm-xxx-xxx	VectorPath	6	
BSF *reg16/32/64*, *mreg16/32/64*	0Fh	BCh	11-xxx-xxx	VectorPath	8/8/9	
BSF *reg16/32/64*, *mem16/32/64*	0Fh	BCh	mm-xxx-xxx	VectorPath	10/11/12	
BSR *reg16/32/64*, *mreg16/32/64*	0Fh	BDh	11-xxx-xxx	VectorPath	11	
BSR *reg16/32/64*, *mem16/32/64*	0Fh	BDh	mm-xxx-xxx	VectorPath	14/13/13	
BSWAP EAX/RAX/R8	0Fh	C8h		DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
BSWAP EBP/RBP/R13	0Fh	CDh		DirectPath	1	
BSWAP EBX/RBX/R11	0Fh	CBh		DirectPath	1	
BSWAP ECX/RCX/R9	0Fh	C9h		DirectPath	1	
BSWAP EDI/RDI/R15	0Fh	CFh		DirectPath	1	
BSWAP EDX/RDX/R10	0Fh	CAh		DirectPath	1	
BSWAP ESI/RSI/R14	0Fh	CEh		DirectPath	1	
BSWAP ESP/RSP/R12	0Fh	CCh		DirectPath	1	
BT *mreg16/32/64*, *reg16/32/64*	0Fh	A3h	11-xxx-xxx	DirectPath	1	
BT *mem16/32/64*, *reg16/32/64*	0Fh	A3h	mm-xxx-xxx	VectorPath	8	
BT *mreg16/32/64*, *imm8*	0Fh	BAh	11-100-xxx	DirectPath	1	
BT *mem16/32/64*, *imm8*	0Fh	BAh	mm-100-xxx	DirectPath	4	
BTC *mreg16/32/64*, *reg16/32/64*	0Fh	BBh	11-xxx-xxx	Double	2	
BTC *mem16/32/64*, *reg16/32/64*	0Fh	BBh	mm-xxx-xxx	VectorPath	9	
BTC *mreg16/32/64*, *imm8*	0Fh	BAh	11-111-xxx	Double	2	
BTC *mem16/32/64*, *imm8*	0Fh	BAh	mm-111-xxx	VectorPath	5	
BTR *mreg16/32/64*, *reg16/32/64*	0Fh	B3h	11-xxx-xxx	Double	2	
BTR *mem16/32/64*, *reg16/32/64*	0Fh	B3h	mm-xxx-xxx	VectorPath	9	
BTR *mreg16/32/64*, *imm8*	0Fh	BAh	11-110-xxx	Double	2	
BTR *mem16/32/64*, *imm8*	0Fh	BAh	mm-110-xxx	VectorPath	5	
BTS *mreg16/32/64*, *reg16/32/64*	0Fh	ABh	11-xxx-xxx	Double	2	
BTS *mem16/32/64*, *reg16/32/64*	0Fh	ABh	mm-xxx-xxx	VectorPath	9	
BTS *mreg16/32/64*, *imm8*	0Fh	BAh	11-101-xxx	Double	2	
BTS *mem16/32/64*, *imm8*	0Fh	BAh	mm-101-xxx	VectorPath	5	
CALL *disp16/32* (near, displacement)	E8h			VectorPath	3	2

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
CALL *mem16/32/64* (near, indirect)	FFh		mm-010-xxx	VectorPath	4	
CALL *mreg16/32/64* (near, indirect)	FFh		11-010-xxx	VectorPath	4	
CALL *mem16:16/32* (far, indirect)	FFh		11-011-xxx	VectorPath	~	
CALL *pntr16:16/32* (far, direct, no CPL change)	9Ah			VectorPath	33	
CALL *pntr16:16/32* (far, direct, CPL change)	9Ah			VectorPath	150	
CBW/CWDE/CDQE	98h			DirectPath	1	
CLC	F8h			DirectPath	1	
CLD	FCh			DirectPath	1	
CLFLUSH	0Fh	AEh	mm-111-xx	DirectPath	~	
CLI	FAh			VectorPath	4	
CLTS	0Fh	06h		VectorPath	10	
CMC	F5h			DirectPath	1	
CMOVA/CMOVNBE *reg16/32/64*, *mem16/32/64*	0Fh	47h	mm-xxx-xxx	DirectPath	4	
CMOVA/CMOVNBE *reg16/32/64*, *reg16/32/64*	0Fh	47h	11-xxx-xxx	DirectPath	1	
CMOVAE/CMOVNB/CMOVNC *reg16/32/64*, *mem16/32/64*	0Fh	43h	mm-xxx-xxx	DirectPath	4	
CMOVAE/CMOVNB/CMOVNC *reg16/32/64*, *reg16/32/64*	0Fh	43h	11-xxx-xxx	DirectPath	1	
CMOVB/CMOVC/CMOVNAE *reg16/32/64*, *mem16/32/64*	0Fh	42h	mm-xxx-xxx	DirectPath	4	
CMOVB/CMOVC/CMOVNAE *reg16/32/64*, *reg16/32/64*	0Fh	42h	11-xxx-xxx	DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
CMOVBE/CMOVNA *reg16/32/64*, *mem16/32/64*	0Fh	46h	mm-xxx-xxx	DirectPath	4	
CMOVBE/CMOVNA *reg16/32/64*, *reg16/32/64*	0Fh	46h	11-xxx-xxx	DirectPath	1	
CMOVE/CMOVZ *reg16/32/64*, *mem16/32/64*	0Fh	44h	mm-xxx-xxx	DirectPath	4	
CMOVE/CMOVZ *reg16/32/64*, *reg16/32/64*	0Fh	44h	11-xxx-xxx	DirectPath	1	
CMOVG/CMOVNLE *reg16/32/64*, *mem16/32/64*	0Fh	4Fh	mm-xxx-xxx	DirectPath	4	
CMOVG/CMOVNLE *reg16/32/64*, *reg16/32/64*	0Fh	4Fh	11-xxx-xxx	DirectPath	1	
CMOVGE/CMOVNL *reg16/32/64*, *mem16/32/64*	0Fh	4Dh	mm-xxx-xxx	DirectPath	4	
CMOVGE/CMOVNL *reg16/32/64*, *reg16/32/64*	0Fh	4Dh	11-xxx-xxx	DirectPath	1	
CMOVL/CMOVNGE *reg16/32/64*, *mem16/32/64*	0Fh	4Ch	mm-xxx-xxx	DirectPath	4	
CMOVL/CMOVNGE *reg16/32/64*, *reg16/32/64*	0Fh	4Ch	11-xxx-xxx	DirectPath	1	
CMOVLE/CMOVNG *reg16/32/64*, *mem16/32/64*	0Fh	4Eh	mm-xxx-xxx	DirectPath	4	
CMOVLE/CMOVNG *reg16/32/64*, *reg16/32/64*	0Fh	4Eh	11-xxx-xxx	DirectPath	1	
CMOVNE/CMOVNZ *reg16/32/64*, *mem16/32/64*	0Fh	45h	mm-xxx-xxx	DirectPath	4	
CMOVNE/CMOVNZ *reg16/32/64*, *reg16/32/64*	0Fh	45h	11-xxx-xxx	DirectPath	1	
CMOVNO *reg16/32/64*, *mem16/32/64*	0Fh	41h	mm-xxx-xxx	DirectPath	4	
CMOVNO *reg16/32/64*, *reg16/32/64*	0Fh	41h	11-xxx-xxx	DirectPath	1	
CMOVNP/CMOVPO *reg16/32/64*, *mem16/32/64*	0Fh	4Bh	mm-xxx-xxx	DirectPath	4	
CMOVNP/CMOVPO *reg16/32/64*, *reg16/32/64*	0Fh	4Bh	11-xxx-xxx	DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
CMOVNS *reg16/32/64*, *mem16/32/64*	0Fh	49h	mm-xxx-xxx	DirectPath	4	
CMOVNS *reg16/32/64*, *reg16/32/64*	0Fh	49h	11-xxx-xxx	DirectPath	1	
CMOVO *reg16/32/64*, *mem16/32/64*	0Fh	40h	mm-xxx-xxx	DirectPath	4	
CMOVO *reg16/32/64*, *reg16/32/64*	0Fh	40h	11-xxx-xxx	DirectPath	1	
CMOVP/CMOVPE *reg16/32/64*, *mem16/32/64*	0Fh	4Ah	mm-xxx-xxx	DirectPath	4	
CMOVP/CMOVPE *reg16/32/64*, *reg16/32/64*	0Fh	4Ah	11-xxx-xxx	DirectPath	1	
CMOVS *reg16/32/64*, *mem16/32/64*	0Fh	48h	mm-xxx-xxx	DirectPath	4	
CMOVS *reg16/32/64*, *reg16/32/64*	0Fh	48h	11-xxx-xxx	DirectPath	1	
CMP *mem8*, *reg8*	38h		mm-xxx-xxx	DirectPath	4	
CMP *mreg8*, *reg8*	38h		11-xxx-xxx	DirectPath	1	
CMP *mem16/32/64*, *reg16/32/64*	39h		mm-xxx-xxx	DirectPath	4	
CMP *mreg16/32/64*, *reg16/32/64*	39h		11-xxx-xxx	DirectPath	1	
CMP *reg8*, *mem8*	3Ah		mm-xxx-xxx	DirectPath	4	
CMP *reg8*, *mreg8*	3Ah		11-xxx-xxx	DirectPath	1	
CMP *reg16/32/64*, *mem16/32/64*	3Bh		mm-xxx-xxx	DirectPath	4	
CMP *reg16/32/64*, *mreg16/32/64*	3Bh		11-xxx-xxx	DirectPath	1	
CMP AL, *imm8*	3Ch			DirectPath	1	
CMP AX/EAX, *imm16/32*	3Dh			DirectPath	1	
CMP RAX, *imm32* (sign extended)	3Dh			DirectPath	1	
CMP *mem8*, *imm8*	80h		mm-111-xxx	DirectPath	4	
CMP *mreg8*, *imm8*	80h		11-111-xxx	DirectPath	1	
CMP *mem16/32/64*, *imm16/32*	81h		mm-111-xxx	DirectPath	4	
CMP *mreg16/32/64*, *imm16/32*	81h		11-111-xxx	DirectPath	1	
CMP *mem16/32/64*, *imm8* (sign extended)	83h		mm-111-xxx	DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
CMP *mreg16/32/64*, *imm8* (sign extended)	83h		11-111-xxx	DirectPath	1	
CMPS *mem8*, *mem8*	A6h			VectorPath	6	7
CMPS *mem16/32/64*, *mem16/32/64*	A7h			VectorPath	6	7
CMPSB	A6h			VectorPath	6	7
CMPSD	A7h			VectorPath	6	7
CMPSQ	A7			VectorPath	6	7
CMPSW	A7h			VectorPath	6	7
CMPXCHG *mem8*, *reg8*	0Fh	B0h	mm-xxx-xxx	VectorPath	5	
CMPXCHG *mreg8*, *reg8*	0Fh	B0h	11-xxx-xxx	VectorPath	3	
CMPXCHG *mem16/32/64*, *reg16/32/64*	0Fh	B1h	mm-xxx-xxx	VectorPath	5	
CMPXCHG *mreg16/32/64*, *reg16/32/64*	0Fh	B1h	11-xxx-xxx	VectorPath	3	
CMPXCHG8B *mem64*	0Fh	C7h	mm-xxx-xxx	VectorPath	10	
CPUID (function 0)	0Fh	A2h		VectorPath	36	
CPUID (function 1)	0Fh	A2h		VectorPath	152	
CPUID (function 2)	0Fh	A2h		VectorPath	38	
CPUID (function 8000_0001h)	0Fh	A2h		VectorPath		
CPUID (function 8000_0002h)	0Fh	A2h		VectorPath		
CPUID (function 8000_0003h)	0Fh	A2h		VectorPath		
CPUID (function 8000_0004h)	0Fh	A2h		VectorPath		
CPUID (function 8000_0007h)	0Fh	A2h		VectorPath		
CPUID (function 8000_0008h)	0Fh	A2h		VectorPath		
CWD/CDQ/CQO	99h			DirectPath	1	
DAA	27h			VectorPath	7	
DAS	2Fh			VectorPath	7	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
DEC AX/EAX	48h			DirectPath	1	9
DEC BP/EBP	4Dh			DirectPath	1	9
DEC BX/EBX	4Bh			DirectPath	1	9
DEC CX/ECX	49h			DirectPath	1	9
DEC DI/EDI	4Fh			DirectPath	1	9
DEC DX/EDX	4Ah			DirectPath	1	9
DEC SI/ESI	4Eh			DirectPath	1	9
DEC SP/ESP	4Ch			DirectPath	1	9
DEC *mem8*	FEh		mm-001-xxx	DirectPath	4	
DEC *mreg8*	FEh		11-001-xxx	DirectPath	1	
DEC *mem16/32/64*	FFh		mm-001-xxx	DirectPath	4	
DEC *mreg16/32/64*	FFh		11-001-xxx	DirectPath	1	
DIV *mem8*	F6h		mm-110-xxx	VectorPath	16	
DIV *mreg8*	F6h		11-110-xxx	VectorPath	16	
DIV *mem16/32/64*	F7h		mm-110-xxx	VectorPath	23/39/71	
DIV *mreg16/32/64*	F7h		11-110-xxx	VectorPath	23/39/71	
ENTER	C8h			VectorPath	14/17/19/21	5
IDIV mreg8	F6h		11-111-xxx	VectorPath	18	
IDIV mem8	F6h		mm-111-xxx	VectorPath	19	
IDIV mreg16/32/64	F7h		11-111-xxx	VectorPath	26/42/74	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
IDIV mem16/32/64	F7h		mm-111-xxx	VectorPath	27/43/75	
IMUL reg16, imm16	69h		11-xxx-xxx	VectorPath	4	
IMUL reg32/64, imm32/(32 sign extended)	69h		11-xxx-xxx	DirectPath	3/4	
IMUL reg16, mreg16, imm16	69h		11-xxx-xxx	VectorPath	4	
IMUL reg32/64, mreg32/64, imm32/(32 sign extended)	69h		11-xxx-xxx	DirectPath	3/4	
IMUL reg16/32/64, mem16/32/64, imm16/32/(32 sign extended)	69h		mm-xxx-xxx	VectorPath	7/7/8	
IMUL reg16/32/64, imm8 (sign extended)	6Bh		11-xxx-xxx	VectorPath	4/3/4	
IMUL reg16/32/64, mreg16/32/64, imm8 (signed)	6Bh		11-xxx-xxx	VectorPath	4/3/4	
IMUL reg16/32/64, mem16/32/64, imm8 (signed)	6Bh		mm-xxx-xxx	VectorPath	7/7/8	
IMUL mreg8	F6h		11-101-xxx	DirectPath	3	
IMUL mem8	F6h		mm-101-xxx	DirectPath	6	
IMUL mreg16	F7h		11-101-xxx	VectorPath	4	
IMUL mreg32/64	F7h		11-101-xxx	Double	3/5	
IMUL mem16	F7h		mm-101-xxx	VectorPath	7	
IMUL mem32/64	F7h		mm-101-xxx	Double	6/8	
IMUL reg16/32/64, mreg16/32/64	0Fh	AFh	11-xxx-xxx	DirectPath	3/3/4	
IMUL reg16/32/64, mem16/32/64	0Fh	AFh	mm-xxx-xxx	DirectPath	6/6/7	
IN AL, imm8	E4h			VectorPath	184	
IN AX, imm8	E5h			VectorPath	184	
IN EAX, imm8	E5h			VectorPath	184	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
IN AL, DX	ECh			VectorPath	179	
IN AX, DX	EDh			VectorPath	179	
IN EAX, DX	EDh			VectorPath	181	
INC AX, EAX	40h			DirectPath	1	9
INC CX, ECX	41h			DirectPath	1	9
INC DX, EDX	42h			DirectPath	1	9
INC BX, EBX	43h			DirectPath	1	9
INC SP, ESP	44h			DirectPath	1	9
INC BP, EBP	45h			DirectPath	1	9
INC SI, ESI	46h			DirectPath	1	9
INC DI, EDI	47h			DirectPath	1	9
INC mreg8	FEh		11-000-xxx	DirectPath	1	
INC mem8	FEh		mm-000-xxx	DirectPath	4	
INC mreg16/32/64	FFh		11-000-xxx	DirectPath	1	
INC mem16/32/64	FFh		mm-000-xxx	DirectPath	4	
INSB/INS mem8, DX	6Ch			VectorPath	184	
INSD/INS mem32, DX	6Dh			VectorPath	185	
INSW/INS mem16, DX	6Dh			VectorPath	186	
INT imm8 (no CPL change)	CDh			VectorPath	87–109	
INT imm8 (CPL change)	CDh			VectorPath	91–112	
INVD	0Fh	08h		VectorPath	247	
INVLPG	0Fh	01h	mm-111-xxx	VectorPath	101/80	8
IRET, IRETD, IRETQ (from 64-bit to 64-bit)	CFh			VectorPath	91	
IRET, IRETD, IRETQ (from 64-bit to 32-bit)	CFh			VectorPath	111	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
JA/JNBE *disp8*	77h			DirectPath	1	1
JA/JNBE *disp16/32*	0Fh	87h		DirectPath	1	1
JAE/JNB/JNC *disp8*	73h			DirectPath	1	1
JAE/JNB/JNC *disp16/32*	0Fh	83h		DirectPath	1	1
JB/JC/JNAE *disp8*	72h			DirectPath	1	1
JB/JC/JNAE *disp16/32*	0Fh	82h		DirectPath	1	1
JBE/JNA *disp8*	76h			DirectPath	1	1
JBE/JNA *disp16/32*	0Fh	86h		DirectPath	1	1
JCXZ/JECXZ/JRCXZ *disp8*	E3h			DirectPath	2	1
JE/JZ *disp8*	74h			DirectPath	1	1
JE/JZ *disp16/32*	0Fh	84h		DirectPath	1	1
JG/JNLE *disp8*	7Fh			DirectPath	1	1
JG/JNLE *disp16/32*	0Fh	8Fh		DirectPath	1	1
JGE/JNL *disp8*	7Dh			DirectPath	1	1
JGE/JNL *disp16/32*	0Fh	8Dh		DirectPath	1	1
JL/JNGE *disp8*	7Ch			DirectPath	1	1
JL/JNGE *disp16/32*	0Fh	8Ch		DirectPath	1	1
JLE/JNG *disp8*	7Eh			DirectPath	1	1
JLE/JNG *disp16/32*	0Fh	8Eh		DirectPath	1	1
JMP *disp8* (short)	EBh			DirectPath	1	
JMP *disp16/32* (near, displacement)	E9h			DirectPath	1	
JMP *mem16/32/64* (near, indirect)	FFh		mm-100-xxx	DirectPath	4	
JMP *mreg16/32/64* (near, indirect)	FFh		11-100-xxx	DirectPath	1	
JMP *mem16:16/32* (far, indirect, no call gate)	FFh		mm-101-xxx	VectorPath	34	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see "32/64-Bit vs. 16-Bit Forms of the LEA Instruction" on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in "Latency of Repeated String Instructions" on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
JMP *mem16:16/32* (far, indirect, call gate)	FFh		mm-101-xxx	VectorPath	123	
JMP *pntr16:16/32* (far, direct, no call gate)	EAh			VectorPath	31	
JMP *pntr16:16/32* (far, direct, call gate)	EAh			VectorPath	120	
JNE/JNZ *disp8*	75h			DirectPath	1	1
JNE/JNZ *disp16/32*	0Fh	85h		DirectPath	1	1
JNO *disp8*	71h			DirectPath	1	1
JNO *disp16/32*	0Fh	81h		DirectPath	1	1
JNP/JPO *disp8*	7Bh			DirectPath	1	1
JNP/JPO *disp16/32*	0Fh	8Bh		DirectPath	1	1
JNS *disp8*	79h			DirectPath	1	1
JNS *disp16/32*	0Fh	89h		DirectPath	1	1
JO *disp8*	70h			DirectPath	1	1
JO *disp16/32*	0Fh	80h		DirectPath	1	1
JP/JPE *disp8*	7Ah			DirectPath	1	1
JP/JPE *disp16/32*	0Fh	8Ah		DirectPath	1	1
JS *disp8*	78h			DirectPath	1	1
JS *disp16/32*	0Fh	88h		DirectPath	1	1
LAHF	9Fh			VectorPath	3	
LAR reg16/32/64, mreg16/32/64	0Fh	02h	11-xxx-xxx	VectorPath	22	
LAR reg16/32/64, mem16/32/64	0Fh	02h	mm-xxx-xxx	VectorPath	24	
LDS reg16/32, mem16:16/32	C5h		mm-xxx-xxx	VectorPath	~	
LEA reg16, mem16/32/64	8Dh		mm-xxx-xxx	VectorPath	3	4
LEA reg32/64, mem16/32/64	8Dh		mm-xxx-xxx	DirectPath	2	4
LEAVE (16 bit stack size)	C9h			VectorPath	3	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see "32/64-Bit vs. 16-Bit Forms of the LEA Instruction" on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in "Latency of Repeated String Instructions" on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
LEAVE (32 or 64 bit stack size)	C9h			Double	3	
LES reg16/32, mem32/48	C4h		mm-xxx-xxx	VectorPath	~	
LFS reg16/32, mem32/48	0Fh	B4h		VectorPath	~	
LGDT mem16:32	0Fh	01h	mm-010-xxx	VectorPath	37	
LGDT mem16:64	0Fh	01h	mm-010-xxx	VectorPath	~	
LGS reg16/32, mem32/48	0Fh	B5h		VectorPath	~	
LIDT mem16:32	0Fh	01h	mm-011-xxx	VectorPath	148	
LIDT mem16:64	0Fh	01h	mm-011-xxx	VectorPath	~	
LLDT mreg16	0Fh	00h	11-010-xxx	VectorPath	34	
LLDT mem16	0Fh	00h	mm-010-xxx	VectorPath	35	
LMSW mreg16	0Fh	01h	11-100-xxx	VectorPath	11	
LMSW mem16	0Fh	01h	mm-100-xxx	VectorPath	12	
LODS/LODSB mem8	ACh			VectorPath	5	7
LODS/LODSW mem16	ADh			VectorPath	5	7
LODS/LODSD mem32	ADh			VectorPath	4	7
LODS/LODSQ mem64	ADh			VectorPath	~	7
LOOP disp8	E2h			VectorPath	9/8	8
LOOPE/LOOPZ disp8	E1h			VectorPath	9/8	8
LOOPNE/LOOPNZ disp8	E0h			VectorPath	9/8	8
LSL reg16/32/64, mreg16/32	0Fh	03h	11-xxx-xxx	VectorPath	21	
LSL reg16/32/64, mem16/32	0Fh	03h	mm-xxx-xxx	VectorPath	23	
LSS reg16/32/64, mem16:16/32	0Fh	B2h	mm-xxx-xxx	VectorPath	~	
LTR mreg16	0Fh	00h	11-011-xxx	VectorPath	~	
LTR mem16	0Fh	00h	mm-011-xxx	VectorPath	~	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see "32/64-Bit vs. 16-Bit Forms of the LEA Instruction" on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in "Latency of Repeated String Instructions" on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
MFENCE	0Fh	AEh	11-110-000	VectorPath	~	
MOV mreg8, reg8	88h		11-xxx-xxx	DirectPath	1	
MOV mem8, reg8	88h		mm-xxx-xxx	DirectPath	3	
MOV mreg16/32/64, reg16/32/64	89h		11-xxx-xxx	DirectPath	1	
MOV mem16/32/64, reg16/32/64	89h		mm-xxx-xxx	DirectPath	3	
MOV reg8, mreg8	8Ah		11-xxx-xxx	DirectPath	1	
MOV reg8, mem8	8Ah		mm-xxx-xxx	DirectPath	4	
MOV reg16/32/64, mreg16/32/64	8Bh		11-xxx-xxx	DirectPath	1	
MOV reg16, mem16	8Bh		mm-xxx-xxx	DirectPath	4	
MOV reg32/64, mem32/64	8Bh		mm-xxx-xxx	DirectPath	3	
MOV mreg16/32/64, sreg	8Ch		11-xxx-xxx	DirectPath	4/3	8
MOV mem16, sreg	8Ch		mm-xxx-xxx	Double	4	
MOV sreg, mreg16/32/64	8Eh		11-xxx-xxx	VectorPath	8	
MOV sreg, mem16	8Eh		mm-xxx-xxx	VectorPath	10	
MOV AL, mem8	A0h			DirectPath	4	
MOV AX/EAX/RAX, mem16/32/64	A1h			DirectPath	4/3/3	
MOV mem8, AL	A2h			DirectPath	3	
MOV mem16/32/64, AX/EAX/RAX	A3h			DirectPath	3	
MOV AL, imm8	B0h			DirectPath	1	
MOV CL, imm8	B1h			DirectPath	1	
MOV DL, imm8	B2h			DirectPath	1	
MOV BL, imm8	B3h			DirectPath	1	
MOV AH, imm8	B4h			DirectPath	1	
MOV CH, imm8	B5h			DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
MOV DH, imm8	B6h			DirectPath	1	
MOV BH, imm8	B7h			DirectPath	1	
MOV AX/EAX/RAX/R8, imm16/32/64	B8h			DirectPath	1	
MOV CX/ECX/RCX/R9, imm16/32/64	B9h			DirectPath	1	
MOV DX/EDX/RDX/R10, imm16/32/64	BAh			DirectPath	1	
MOV BX/EBX/RBX/R11, imm16/32/64	BBh			DirectPath	1	
MOV SP/ESP/RSP/R12, imm16/32/64	BCh			DirectPath	1	
MOV BP/EBP/RBP/R13, imm16/32/64	BDh			DirectPath	1	
MOV SI/ESI/RSI/R14, imm16/32/64	BEh			DirectPath	1	
MOV DI/EDI/RDI/R15, imm16/32/64	BFh			DirectPath	1	
MOV mreg8, imm8	C6h		11-000-xxx	DirectPath	1	
MOV mem8, imm8	C6h		mm-000-xxx	DirectPath	3	
MOV mreg16/32/64, imm16/32	C7h		11-000-xxx	DirectPath	1	
MOV mem16/32/64, imm16/32	C7h		mm-000-xxx	DirectPath	3	
MOVSB/MOVS mem8, mem8	A4h			VectorPath	5	7
MOVSD/MOVS mem16, mem16	A5h			VectorPath	5	7
MOVSW/MOVS mem32, mem32	A5h			VectorPath	5	7
MOVSQ/MOVS mem64, mem64	A5h			VectorPath	~	7
MOVSX reg16/32/64, mreg8	0Fh	BEh	11-xxx-xxx	DirectPath	1	
MOVSX reg16/32/64, mem8	0Fh	BEh	mm-xxx-xxx	DirectPath	4	
MOVSX reg32/64, mreg16	0Fh	BFh	11-xxx-xxx	DirectPath	1	
MOVSX reg32/64, mem16	0Fh	BFh	mm-xxx-xxx	DirectPath	4	
MOVSXD reg64, mreg32	63h			DirectPath	1	
MOVSXD reg64, mem32	63h			DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
MOVZX reg16/32/64, mreg8	0Fh	B6h	11-xxx-xxx	DirectPath	1	
MOVZX reg16/32/64, mem8	0Fh	B6h	mm-xxx-xxx	DirectPath	4	
MOVZX reg32/64, mreg16	0Fh	B7h	11-xxx-xxx	DirectPath	1	
MOVZX reg32/64, mem16	0Fh	B7h	mm-xxx-xxx	DirectPath	4	
MUL mreg8	F6h		11-100-xxx	DirectPath	3	
MUL AL, mem8	F6h		mm-100-xx	DirectPath	6	
MUL mreg16	F7h		11-100-xxx	VectorPath	4	
MUL mem16	F7h		mm-100-xxx	VectorPath	7	
MUL mreg32	F7h		11-100-xxx	Double	3	
MUL mem32	F7h		mm-100-xx	Double	6	
MUL mreg64	F7h		11-100-xxx	Double	5	
MUL mem64	F7h		mm-100-xx	Double	8	
NEG mreg8	F6h		11-011-xxx	DirectPath	1	
NEG mem8	F6h		mm-011-xxx	DirectPath	4	
NEG mreg16/32/64	F7h		11-011-xxx	DirectPath	1	
NEG mem16/32/64	F7h		mm-011-xx	DirectPath	4	
NOP (XCHG EAX, EAX)	90h			DirectPath	~0	6
NOT mreg8	F6h		11-010-xxx	DirectPath	1	
NOT mem8	F6h		mm-010-xx	DirectPath	4	
NOT mreg16/32/64	F7h		11-010-xxx	DirectPath	1	
NOT mem16/32/64	F7h		mm-010-xx	DirectPath	4	
OR *mreg8*, *reg8*	08h		11-xxx-xxx	DirectPath	1	
OR *mem8*, *reg8*	08h		mm-xxx-xxx	DirectPath	4	
OR *mreg16/32/64*, *reg16/32/64*	09h		11-xxx-xxx	DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
OR *mem16/32/64*, *reg16/32/64*	09h		mm-xxx-xxx	DirectPath	4	
OR *reg8*, *mreg8*	0Ah		11-xxx-xxx	DirectPath	1	
OR *reg8*, *mem8*	0Ah		mm-xxx-xxx	DirectPath	4	
OR *reg16/32/64*, *mreg16/32/64*	0Bh		11-xxx-xxx	DirectPath	1	
OR *reg16/32/64*, *mem16/32/64*	0Bh		mm-xxx-xxx	DirectPath	4	
OR AL, *imm8*	0Ch			DirectPath	1	
OR AX, *imm16*	0Dh			DirectPath	1	
OR EAX, *imm32*	0Dh			DirectPath	1	
OR RAX, *imm32* (sign extended)	0Dh			DirectPath	1	
OR *mreg8*, *imm8*	80h		11-001-xxx	DirectPath	1	
OR *mem8*, *imm8*	80h		mm-001-xxx	DirectPath	4	
OR *mreg16/32/64*, *imm16/32*	81h		11-001-xxx	DirectPath	1	
OR *mem16/32/64*, *imm16/32*	81h		mm-001-xxx	DirectPath	4	
OR *mreg16/32/64*, *imm8* (sign extended)	83h		11-001-xxx	DirectPath	1	
OR *mem16/32/64*, *imm8* (sign extended)	83h		mm-001-xxx	DirectPath	4	
OUT imm8, AL	E6h			VectorPath	~	
OUT imm8, AX	E7h			VectorPath	~	
OUT imm8, EAX	E7h			VectorPath	~	
OUT DX, AL	EEh			VectorPath	165	
OUT DX, AX	EFh			VectorPath	165	
OUT DX, EAX	EFh			VectorPath	165	
POP ES	07h			VectorPath	10	
POP SS	17h			VectorPath	31	
POP DS	1Fh			VectorPath	10	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
POP FS	0Fh	A1h		VectorPath	10	
POP GS	0Fh	A9h		VectorPath	10	
POP AX/EAX/RAX/(R8)	58h			Double	3	
POP CX/ECX/RCX/(R9)	59h			Double	3	
POP DX/EDX/RDX/(R10)	5Ah			Double	3	
POP BX/EBX/RBX/(R11)	5Bh			Double	3	
POP SP/ESP/RSP/(R12)	5Ch			Double	3	
POP BP/EBP/RBP/(R13)	5Dh			Double	3	
POP SI/ESI/RSI/(R14)	5Eh			Double	3	
POP DI/EDI/RDI/(R15)	5Fh			Double	3	
POP mreg 16/32/64	8Fh		11-000-xxx	VectorPath	3	
POP mem 16/32/64	8Fh		mm-000-xxx	VectorPath	3	
POPA/POPAD	61h			VectorPath	6	
POPF/POPFD/POPFQ	9Dh			VectorPath	15	
PUSH ES	06h			VectorPath	3	2
PUSH CS	0Eh			VectorPath	3	
PUSH FS	0Fh	A0h		VectorPath	3	
PUSH GS	0Fh	A8h		VectorPath	3	
PUSH SS	16h			VectorPath	3	
PUSH DS	1Eh			VectorPath	3	2
PUSH AX/EAX/RAX/(R8)	50h			DirectPath	3	2
PUSH CX/ECX/RCX/(R9)	51h			DirectPath	3	2
PUSH DX/EDX/RDX/(R10)	52h			DirectPath	3	2
PUSH BX/EBX/RBX/(R11)	53h			DirectPath	3	2

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
PUSH SP/ESP/RSP/(R12)	54h			DirectPath	3	2
PUSH BP/EBP/RBP/(R13)	55h			DirectPath	3	2
PUSH SI/ESI/RSI/(R14)	56h			DirectPath	3	2
PUSH DI/EDI/RDI/(R15)	57h			DirectPath	3	2
PUSH imm8	6Ah			DirectPath	3	2
PUSH imm16/32	68h			DirectPath	3	2
PUSH mreg16/32/64	FFh		11-110-xxx	DirectPath	3	
PUSH mem16/32/64	FFh		mm-110-xxx	Double	3	2
PUSHA/PUSHAD	60h			VectorPath	6	
PUSHF/PUSHFD/PUSHFQ	9Ch			VectorPath	4	
RCL mreg8, imm8	C0h		11-010-xxx	VectorPath	7	
RCL mem8, imm8	C0h		mm-010-xxx	VectorPath	8	
RCL mreg16/32/64, imm8	C1h		11-010-xxx	VectorPath	7	
RCL mem16/32/64, imm8	C1h		mm-010-xxx	VectorPath	8	
RCL mreg8, 1	D0h		11-010-xxx	DirectPath	1	
RCL mem8, 1	D0h		mm-010-xxx	DirectPath	4	
RCL mreg16/32/64, 1	D1h		11-010-xxx	DirectPath	1	
RCL mem16/32/64, 1	D1h		mm-010-xxx	DirectPath	4	
RCL mreg8, CL	D2h		11-010-xxx	VectorPath	6	
RCL mem8, CL	D2h		mm-010-xxx	VectorPath	7	
RCL mreg16/32/64, CL	D3h		11-010-xxx	VectorPath	6	
RCL mem16/32/64, CL	D3h		mm-010-xxx	VectorPath	7	
RCR mreg8, imm8	C0h		11-011-xxx	VectorPath	5	
RCR mem8, imm8	C0h		mm-011-xxx	VectorPath	6	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
RCR mreg16/32/64, imm8	C1h		11-011-xxx	VectorPath	5	
RCR mem16/32/64, imm8	C1h		mm-011-xxx	VectorPath	6	
RCR mreg8, 1	D0h		11-011-xxx	DirectPath	1	
RCR mem8, 1	D0h		mm-011-xxx	DirectPath	4	
RCR mreg16/32/64, 1	D1h		11-011-xxx	DirectPath	1	
RCR mem16/32/64, 1	D1h		mm-011-xxx	DirectPath	4	
RCR mreg8, CL	D2h		11-011-xxx	VectorPath	4	
RCR mem8, CL	D2h		mm-011-xxx	VectorPath	6	
RCR mreg16/32/64, CL	D3h		11-011-xxx	VectorPath	4	
RCR mem16/32/64, CL	D3h		mm-011-xxx	VectorPath	6	
RDMSR	0Fh	32h		VectorPath	87	
RDPMC	0Fh	33h		VectorPath	~	
RDTSC	0Fh	31h		VectorPath	12	
RET near imm16	C2h			VectorPath	5	
RET near	C3h			Double	5	
RET far imm16 (no CPL change)	CAh			VectorPath	31–44	
RET far imm16 (CPL change)	CAh			VectorPath	57-72	
RET far (no CPL change)	CBh			VectorPath	31–44	
RET far (CPL change)	CBh			VectorPath	57-72	
ROL mreg8, imm8	C0h		11-000-xxx	DirectPath	1	3
ROL mem8, imm8	C0h		mm-000-xxx	DirectPath	4	3
ROL mreg16/32/64, imm8	C1h		11-000-xxx	DirectPath	1	3
ROL mem16/32/64, imm8	C1h		mm-000-xxx	DirectPath	4	3
ROL mreg8, 1	D0h		11-000-xxx	DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
ROL mem8, 1	D0h		mm-000-xxx	DirectPath	4	
ROL mreg16/32/64, 1	D1h		11-000-xxx	DirectPath	1	
ROL mem16/32/64, 1	D1h		mm-000-xxx	DirectPath	4	
ROL mreg8, CL	D2h		11-000-xxx	DirectPath	1	3
ROL mem8, CL	D2h		mm-000-xxx	DirectPath	4	3
ROL mreg16/32/64, CL	D3h		11-000-xxx	DirectPath	1	3
ROL mem16/32/64, CL	D3h		mm-000-xxx	DirectPath	4	3
ROR mreg8, imm8	C0h		11-001-xxx	DirectPath	1	3
ROR mem8, imm8	C0h		mm-001-xxx	DirectPath	4	3
ROR mreg16/32/64, imm8	C1h		11-001-xxx	DirectPath	1	3
ROR mem16/32/64, imm8	C1h		mm-001-xxx	DirectPath	4	3
ROR mreg8, 1	D0h		11-001-xxx	DirectPath	1	
ROR mem8, 1	D0h		mm-001-xxx	DirectPath	4	
ROR mreg16/32/64, 1	D1h		11-001-xxx	DirectPath	1	
ROR mem16/32/64, 1	D1h		mm-001-xxx	DirectPath	4	
ROR mreg8, CL	D2h		11-001-xxx	DirectPath	1	3
ROR mem8, CL	D2h		mm-001-xxx	DirectPath	4	3
ROR mreg16/32/64, CL	D3h		11-001-xxx	DirectPath	1	3
ROR mem16/32/64, CL	D3h		mm-001-xxx	DirectPath	4	3
SAHF	9Eh			DirectPath	1	
SAR mreg8, imm8	C0h		11-111-xxx	DirectPath	1	3
SAR mem8, imm8	C0h		mm-111-xxx	DirectPath	4	3
SAR mreg16/32/64, imm8	C1h		11-111-xxx	DirectPath	1	3
SAR mem16/32/64, imm8	C1h		mm-111-xxx	DirectPath	4	3

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
SAR mreg8, 1	D0h		11-111-xxx	DirectPath	1	
SAR mem8, 1	D0h		mm-111-xxx	DirectPath	4	
SAR mreg16/32/64, 1	D1h		11-111-xxx	DirectPath	1	
SAR mem16/32/64, 1	D1h		mm-111-xxx	DirectPath	4	
SAR mreg8, CL	D2h		11-111-xxx	DirectPath	1	3
SAR mem8, CL	D2h		mm-111-xxx	DirectPath	4	3
SAR mreg16/32/64, CL	D3h		11-111-xxx	DirectPath	1	3
SAR mem16/32/64, CL	D3h		mm-111-xxx	DirectPath	4	3
SBB *mreg8*, *reg8*	18h		11-xxx-xxx	DirectPath	1	
SBB *mem8*, *reg8*	18h		mm-xxx-xxx	DirectPath	4	
SBB *mreg16/32/64*, *reg16/32/64*	19h		11-xxx-xxx	DirectPath	1	
SBB *mem16/32/64*, *reg16/32/64*	19h		mm-xxx-xxx	DirectPath	4	
SBB *reg8*, *mreg8*	1Ah		11-xxx-xxx	DirectPath	1	
SBB *reg8*, *mem8*	1Ah		mm-xxx-xxx	DirectPath	4	
SBB *reg16/32/64*, *mreg16/32/64*	1Bh		11-xxx-xxx	DirectPath	1	
SBB *reg16/32/64*, *mem16/32/64*	1Bh		mm-xxx-xxx	DirectPath	4	
SBB AL, *imm8*	1Ch			DirectPath	1	
SBB AX, *imm16*	1Dh			DirectPath	1	
SBB EAX, *imm32*	1Dh			DirectPath	1	
SBB RAX, *imm32* (sign extended)	1Dh			DirectPath	1	
SBB *mreg8*, *imm8*	80h		11-011-xxx	DirectPath	1	
SBB *mem8*, *imm8*	80h		mm-011-xxx	DirectPath	4	
SBB *mreg16/32/64*, *imm16/32*	81h		11-011-xxx	DirectPath	1	
SBB *mem16/32/64*, *imm16/32*	81h		mm-011-xxx	DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
SBB *mreg16/32/64*, *imm8* (sign extended)	83h		11-011-xxx	DirectPath	1	
SBB *mem16/32/64*, *imm8* (sign extended)	83h		mm-011-xxx	DirectPath	4	
SCASB/SCAS mem8	AEh			VectorPath	4	7
SCASD/SCAS mem32	AFh			VectorPath	4	7
SCASQ/SCAS mem64	AFh			VectorPath	4	7
SCASW/SCAS mem16	AFh			VectorPath	4	7
SETA/SETNBE *mem8*	0Fh	97h	mm-xxx-xxx	DirectPath	3	
SETA/SETNBE *mreg8*	0Fh	97h	11-xxx-xxx	DirectPath	1	
SETAE/SETNB/SETNC *mem8*	0Fh	93h	mm-xxx-xxx	DirectPath	3	
SETAE/SETNB/SETNC *mreg8*	0Fh	93h	11-xxx-xxx	DirectPath	1	
SETB/SETC/SETNAE *mem8*	0Fh	92h	mm-xxx-xxx	DirectPath	3	
SETB/SETC/SETNAE *mreg8*	0Fh	92h	11-xxx-xxx	DirectPath	1	
SETBE/SETNA *mem8*	0Fh	96h	mm-xxx-xxx	DirectPath	3	
SETBE/SETNA *mreg8*	0Fh	96h	11-xxx-xxx	DirectPath	1	
SETE/SETZ *mem8*	0Fh	94h	mm-xxx-xxx	DirectPath	3	
SETE/SETZ *mreg8*	0Fh	94h	11-xxx-xxx	DirectPath	1	
SETG/SETNLE *mem8*	0Fh	9Fh	mm-xxx-xxx	DirectPath	3	
SETG/SETNLE *mreg8*	0Fh	9Fh	11-xxx-xxx	DirectPath	1	
SETGE/SETNL *mem8*	0Fh	9Dh	mm-xxx-xxx	DirectPath	3	
SETGE/SETNL *mreg8*	0Fh	9Dh	11-xxx-xxx	DirectPath	1	
SETL/SETNGE *mem8*	0Fh	9Ch	mm-xxx-xxx	DirectPath	3	
SETL/SETNGE *mreg8*	0Fh	9Ch	11-xxx-xxx	DirectPath	1	
SETLE/SETNG *mem8*	0Fh	9Eh	mm-xxx-xxx	DirectPath	3	
SETLE/SETNG *mreg8*	0Fh	9Eh	11-xxx-xxx	DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
SETNE/SETNZ *mem8*	0Fh	95h	mm-xxx-xxx	DirectPath	3	
SETNE/SETNZ *mreg8*	0Fh	95h	11-xxx-xxx	DirectPath	1	
SETNO *mem8*	0Fh	91h	mm-xxx-xxx	DirectPath	3	
SETNO *mreg8*	0Fh	91h	11-xxx-xxx	DirectPath	1	
SETNP/SETPO *mem8*	0Fh	9Bh	mm-xxx-xxx	DirectPath	3	
SETNP/SETPO *mreg8*	0Fh	9Bh	11-xxx-xxx	DirectPath	1	
SETNS *mem8*	0Fh	99h	mm-xxx-xxx	DirectPath	3	
SETNS *mreg8*	0Fh	99h	11-xxx-xxx	DirectPath	1	
SETO *mem8*	0Fh	90h	mm-xxx-xxx	DirectPath	3	
SETO *mreg8*	0Fh	90h	11-xxx-xxx	DirectPath	1	
SETP/SETPE *mem8*	0Fh	9Ah	mm-xxx-xxx	DirectPath	3	
SETP/SETPE *mreg8*	0Fh	9Ah	11-xxx-xxx	DirectPath	1	
SETS *mem8*	0Fh	98h	mm-xxx-xxx	DirectPath	3	
SETS *mreg8*	0Fh	98h	11-xxx-xxx	DirectPath	1	
SGDT mem48	0Fh	01h	mm-000-xxx	VectorPath	17/18	8
SIDT mem48	0Fh	01h	mm-001-xxx	VectorPath	17/18	8
SHL/SAL mreg8, imm8	C0h		11-100-xxx	DirectPath	1	3
SHL/SAL mem8, imm8	C0h		mm-100-xxx	DirectPath	4	3
SHL/SAL mreg16/32/64, imm8	C1h		11-100-xxx	DirectPath	1	3
SHL/SAL mem16/32/64, imm8	C1h		mm-100-xxx	DirectPath	4	3
SHL/SAL mreg8, 1	D0h		11-100-xxx	DirectPath	1	
SHL/SAL mem8, 1	D0h		mm-100-xxx	DirectPath	4	
SHL/SAL mreg16/32/64, 1	D1h		11-100-xxx	DirectPath	1	
SHL/SAL mem16/32/64, 1	D1h		mm-100-xxx	DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
SHL/SAL mreg8, CL	D2h		11-100-xxx	DirectPath	1	3
SHL/SAL mem8, CL	D2h		mm-100-xxx	DirectPath	4	3
SHL/SAL mreg16/32/64, CL	D3h		11-100-xxx	DirectPath	1	3
SHL/SAL mem16/32/64, CL	D3h		mm-100-xxx	DirectPath	4	3
SHLD mreg16/32/64, reg16/32/64, imm8	0Fh	A4h	11-xxx-xxx	VectorPath	4	3
SHLD mem16/32/64, reg16/32/64, imm8	0Fh	A4h	mm-xxx-xxx	VectorPath	6	3
SHLD mreg16/32/64, reg16/32/64, CL	0Fh	A5h	11-xxx-xxx	VectorPath	4	3
SHLD mem16/32/64, reg16/32/64, CL	0Fh	A5h	mm-xxx-xxx	VectorPath	6	3
SHR mreg8, imm8	C0h		11-101-xxx	DirectPath	1	3
SHR mem8, imm8	C0h		mm-101-xxx	DirectPath	4	3
SHR mreg16/32/64, imm8	C1h		11-101-xxx	DirectPath	1	3
SHR mem16/32/64, imm8	C1h		mm-101-xxx	DirectPath	4	3
SHR mreg8, 1	D0h		11-101-xxx	DirectPath	1	
SHR mem8, 1	D0h		mm-101-xxx	DirectPath	4	
SHR mreg16/32/64, 1	D1h		11-101-xxx	DirectPath	1	
SHR mem16/32/64, 1	D1h		mm-101-xxx	DirectPath	4	
SHR mreg8, CL	D2h		11-101-xxx	DirectPath	1	3
SHR mem8, CL	D2h		mm-101-xxx	DirectPath	4	3
SHR mreg16/32/64, CL	D3h		11-101-xxx	DirectPath	1	3
SHR mem16/32/64, CL	D3h		mm-101-xxx	DirectPath	4	3
SHRD mreg16/32/64, reg16/32/64, imm8	0Fh	ACh	11-xxx-xxx	VectorPath	4	3
SHRD mem16/32/64, reg16/32/64, imm8	0Fh	ACh	mm-xxx-xxx	VectorPath	6	3
SHRD mreg16/32/64, reg16/32/64, CL	0Fh	ADh	11-xxx-xxx	VectorPath	4	3
SHRD mem16/32/64, reg16/32/64, CL	0Fh	ADh	mm-xxx-xxx	VectorPath	6	3

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
SLDT mreg16/32/64	0Fh	00h	11-000-xxx	VectorPath	5	
SLDT mem16/32/64	0Fh	00h	mm-000-xxx	VectorPath	5	
SMSW mreg16/32/64	0Fh	01h	11-100-xxx	VectorPath	4	
SMSW mem16	0Fh	01h	mm-100-xxx	VectorPath	3	
STC	F9h			DirectPath	1	
STD	FDh			Double	2	
STI	FBh			VectorPath	4	
STOSB/STOS mem8	AAh			VectorPath	4	7
STOSW/STOS mem16	ABh			VectorPath	4	7
STOSD/STOS mem32	ABh			VectorPath	4	7
STOSQ/STOS mem64	ABh			VectorPath	4	7
STR mreg16/32/64	0Fh	00h	11-001-xxx	VectorPath	5	
STR mem16	0Fh	00h	mm-001-xxx	VectorPath	5	
SUB *mreg8*, *reg8*	28h		11-xxx-xxx	DirectPath	1	
SUB *mem8*, *reg8*	28h		mm-xxx-xxx	DirectPath	4	
SUB *mreg16/32/64*, *reg16/32/64*	29h		11-xxx-xxx	DirectPath	1	
SUB *mem16/32/64*, *reg16/32/64*	29h		mm-xxx-xxx	DirectPath	4	
SUB *reg8*, *mreg8*	2Ah		11-xxx-xxx	DirectPath	1	
SUB *reg8*, *mem8*	2Ah		mm-xxx-xxx	DirectPath	4	
SUB *reg16/32/64*, *mreg16/32/64*	2Bh		11-xxx-xxx	DirectPath	1	
SUB *reg16/32/64*, *mem16/32/64*	2Bh		mm-xxx-xxx	DirectPath	4	
SUB AL, *imm8*	2Ch			DirectPath	1	
SUB AX, *imm16*	2Dh			DirectPath	1	
SUB EAX, *imm32*	2Dh			DirectPath	1	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
SUB RAX, *imm32* (sign extended)	2Dh			DirectPath	1	
SUB *mreg8*, *imm8*	80h		11-101-xxx	DirectPath	1	
SUB *mem8*, *imm8*	80h		mm-101-xxx	DirectPath	4	
SUB *mreg16/32/64*, *imm16/32*	81h		11-101-xxx	DirectPath	1	
SUB *mem16/32/64*, *imm16/32*	81h		mm-101-xxx	DirectPath	4	
SUB *mreg16/32/64*, *imm8* (sign extended)	83h		11-101-xxx	DirectPath	1	
SUB *mem16/32/64*, *imm8* (sign extended)	83h		mm-101-xxx	DirectPath	4	
SYSCALL	0Fh	05h		VectorPath	27	
SYSENTER	0Fh	34h		VectorPath	~	
SYSEXIT	0Fh	35h		VectorPath	~	
SYSRET	0Fh	07h		VectorPath	35	
TEST mreg8, reg8	84h		11-xxx-xxx	DirectPath	1	
TEST mem8, reg8	84h		mm-xxx-xxx	DirectPath	4	
TEST mreg16/32/64, reg16/32/64	85h		11-xxx-xxx	DirectPath	1	
TEST mem16/32/64, reg16/32/64	85h		mm-xxx-xxx	DirectPath	4	
TEST AL, imm8	A8h			DirectPath	1	
TEST AX/EAX/RAX, imm16/32	A9h			DirectPath	1	
TEST mreg8, imm8	F6h		11-000-xxx	DirectPath	1	
TEST mem8, imm8	F6h		mm-000-xxx	DirectPath	4	
TEST mreg16/32/64, imm16/32	F7h		11-000-xxx	DirectPath	1	
TEST mem16/32/64, imm16/32	F7h		mm-000-xxx	DirectPath	4	
VERR mreg16	0Fh	00h	11-100-xxx	VectorPath	11	
VERR mem16	0Fh	00h	mm-100-xxx	VectorPath	11	
VERW mreg16	0Fh	00h	11-101-xxx	VectorPath	11	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
VERW mem16	0Fh	00h	mm-101-xxx	VectorPath	11	
WAIT	9Bh			DirectPath	~0	6
WBINVD	0Fh	09h		VectorPath	9796/ 9474	8
WRMSR	0Fh	30h		VectorPath	134	
XADD mreg8, reg8	0Fh	C0h	11-100-xxx	VectorPath	2	
XADD mem8, reg8	0Fh	C0h	mm-100-xxx	VectorPath	5	
XADD mreg16/32/64, reg16/32/64	0Fh	C1h	11-101-xxx	VectorPath	2	
XADD mem16/32/64, reg16/32/64	0Fh	C1h	mm-101-xxx	VectorPath	5	
XCHG reg8, mreg8	86h		11-xxx-xxx	VectorPath	2	
XCHG mreg8, reg8	86h		11-xxx-xxx	VectorPath	2	
XCHG reg8, mem8	86h		mm-xxx-xxx	VectorPath	16	
XCHG mem8, reg8	86h		mm-xxx-xxx	VectorPath	16	
XCHG reg16/32/64, mreg16/32/64	87h		11-xxx-xxx	VectorPath	2	
XCHG mreg16/32/64, reg16/32/64	87h		11-xxx-xxx	VectorPath	2	
XCHG reg16/32/64, mem16/32/64	87h		mm-xxx-xxx	VectorPath	16	
XCHG mem16/32/64, reg16/32/64	87h		mm-xxx-xxx	VectorPath	16	
XCHG AX/EAX/RAX, AX/EAX/RAX/(R8) (NOP)	90h			DirectPath	~0	6
XCHG AX/EAX/RAX, CX/ECX/RCX/(R9)	91h			VectorPath	2	
XCHG AX/EAX/RAX, DX/EDX/RDX/(R10)	92h			VectorPath	2	
XCHG AX/EAX/RAX, BX/EBX/RBX/(R11)	93h			VectorPath	2	
XCHG AX/EAX/RAX, SP/ESP/RSP/(R12)	94h			VectorPath	2	
XCHG AX/EAX/RAX, BP/EBP/RBP/(R13)	95h			VectorPath	2	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see “32/64-Bit vs. 16-Bit Forms of the LEA Instruction” on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

Table 15. Integer Instructions (Continued)

Syntax	Encoding			Decode type	Latency	Note
	First byte	Second byte	ModRM byte			
XCHG AX/EAX/RAX, SI/ESI/RSI/(R14)	96h			VectorPath	2	
XCHG AX/EAX/RAX, DI/EDI/RDI/(R15)	97h			VectorPath	2	
XLATB/XLAT mem8	D7h			VectorPath	5	
XOR *mreg8*, *reg8*	30h		11-xxx-xxx	DirectPath	1	
XOR *mem8*, *reg8*	30h		mm-xxx-xxx	DirectPath	4	
XOR *mreg16/32/64*, *reg16/32/64*	31h		11-xxx-xxx	DirectPath	1	
XOR *mem16/32/64*, *reg16/32/64*	31h		mm-xxx-xxx	DirectPath	4	
XOR *reg8*, *mreg8*	32h		11-xxx-xxx	DirectPath	1	
XOR *reg8*, *mem8*	32h		mm-xxx-xxx	DirectPath	4	
XOR *reg16/32/64*, *mreg16/32/64*	33h		11-xxx-xxx	DirectPath	1	
XOR *reg16/32/64*, *mem16/32/64*	33h		mm-xxx-xxx	DirectPath	4	
XOR AL, *imm8*	34h			DirectPath	1	
XOR AX, *imm16*	35h			DirectPath	1	
XOR EAX, *imm32*	35h			DirectPath	1	
XOR RAX, *imm32* (sign extended)	35h			DirectPath	1	
XOR *mreg8*, *imm8*	80h		11-110-xxx	DirectPath	1	
XOR *mem8*, v	80h		mm-110-xxx	DirectPath	4	
XOR *mreg16/32/64*, *imm16/32*	81h		11-110-xxx	DirectPath	1	
XOR *mem16/32/64*, *imm16/32*	81h		mm-110-xxx	DirectPath	4	
XOR *mreg16/32/64*, *imm8* (sign extended)	83h		11-110-xxx	DirectPath	1	
XOR *mem16/32/64*, *imm8* (sign extended)	83h		mm-110-xxx	DirectPath	4	

Notes:

1. *Static timing assumes a predicted branch.*
2. *Store operation also updates ESP—the new register value is available one clock earlier than the specified latency.*
3. *The clock count, regardless of the number of shifts or rotates, as determined by CL or imm8.*
4. *The execution latency of the LEA instruction is dependent on the width of the destination operand. For more information on the use of this instruction, see "32/64-Bit vs. 16-Bit Forms of the LEA Instruction" on page 75.*
5. *Execution latencies for nesting levels 0/1/2/3.*
6. *These instructions have an effective latency as shown. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.*
7. *The latency of repeated string instructions can be found in "Latency of Repeated String Instructions" on page 167.*
8. *The first latency value is for 32-bit mode. The second is for 64-bit mode.*
9. *This opcode is used as a REX prefix in 64-bit mode.*

C.3 MMX™ Technology Instructions

Table 16. MMX™ Technology Instructions

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	ModRM byte				
EMMS	0Fh	77h		DirectPath	FADD/FMUL/ FSTORE	6	2
MOVD mmreg, reg32	0Fh	6Eh	11-xxx-xxx	Double	-	9	1
MOVD mmreg, reg64	0Fh	6Eh	11-xxx-xxx	Double	-	9	1
MOVD mmreg, mem32	0Fh	6Eh	mm-xxx-xxx	DirectPath	FADD/FMUL/ FSTORE	4	2
MOVD mmreg, mem64	0Fh	6Eh	mm-xxx-xxx	DirectPath	FADD/FMUL/ FSTORE	4	2
MOVD reg32, mmreg	0Fh	7Eh	11-xxx-xxx	VectorPath	-	4	1
MOVD reg64, mmreg	0Fh	7Eh	11-xxx-xxx	VectorPath	-	4	1
MOVD mem32, mmreg	0Fh	7Eh	mm-xxx-xxx	DirectPath	FSTORE	2	
MOVD mem64, mmreg	0Fh	7Eh	mm-xxx-xxx	DirectPath	FSTORE	2	
MOVQ mmreg1, mmreg2	0Fh	6Fh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
MOVQ mmreg, mem64	0Fh	6Fh	mm-xxx-xxx	DirectPath	FADD/FMUL/ FSTORE	4	2
MOVQ mmreg2, mmreg1	0Fh	7Fh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
MOVQ mem64, mmreg	0Fh	7Fh	mm-xxx-xxx	DirectPath	FSTORE	2	
PACKSSDW mmreg1, mmreg2	0Fh	6Bh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PACKSSDW mmreg, mem64	0Fh	6Bh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PACKSSWB mmreg1, mmreg2	0Fh	63h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PACKSSWB mmreg, mem64	0Fh	63h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PACKUSWB mmreg1, mmreg2	0Fh	67h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PACKUSWB mmreg, mem64	0Fh	67h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PADDB mmreg1, mmreg2	0Fh	FCh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDB mmreg, mem64	0Fh	FCh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PADDD mmreg1, mmreg2	0Fh	FEh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDD mmreg, mem64	0Fh	FEh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PADDSB mmreg1, mmreg2	0Fh	ECh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDSB mmreg, mem64	0Fh	ECh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	

Notes:

1. *Bits 2, 1, and 0 of the ModRM byte select the integer register.*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*

Table 16. MMX™ Technology Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	ModRM byte				
PADDSW mmreg1, mmreg2	0Fh	EDh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDSW mmreg, mem64	0Fh	EDh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PADDUSB mmreg1, mmreg2	0Fh	DCh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDUSB mmreg, mem64	0Fh	DCh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PADDUSW mmreg1, mmreg2	0Fh	DDh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDUSW mmreg, mem64	0Fh	DDh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PADDW mmreg1, mmreg2	0Fh	FDh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PADDW mmreg, mem64	0Fh	FDh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PAND mmreg1, mmreg2	0Fh	DBh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PAND mmreg, mem64	0Fh	DBh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PANDN mmreg1, mmreg2	0Fh	DFh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PANDN mmreg, mem64	0Fh	DFh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PCMPEQB mmreg1, mmreg2	0Fh	74h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PCMPEQB mmreg, mem64	0Fh	74h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PCMPEQD mmreg1, mmreg2	0Fh	76h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PCMPEQD mmreg, mem64	0Fh	76h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PCMPEQW mmreg1, mmreg2	0Fh	75h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PCMPEQW mmreg, mem64	0Fh	75h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PCMPGTB mmreg1, mmreg2	0Fh	64h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PCMPGTB mmreg, mem64	0Fh	64h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PCMPGTD mmreg1, mmreg2	0Fh	66h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PCMPGTD mmreg, mem64	0Fh	66h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PCMPGTW mmreg1, mmreg2	0Fh	65h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PCMPGTW mmreg, mem64	0Fh	65h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PMADDWD mmreg1, mmreg2	0Fh	F5h	11-xxx-xxx	DirectPath	FMUL	3	
PMADDWD mmreg, mem64	0Fh	F5h	mm-xxx-xxx	DirectPath	FMUL	5	
PMULHW mmreg1, mmreg2	0Fh	E5h	11-xxx-xxx	DirectPath	FMUL	3	
PMULHW mmreg, mem64	0Fh	E5h	mm-xxx-xxx	DirectPath	FMUL	5	
PMULLW mmreg1, mmreg2	0Fh	D5h	11-xxx-xxx	DirectPath	FMUL	3	
PMULLW mmreg, mem64	0Fh	D5h	mm-xxx-xxx	DirectPath	FMUL	5	

Notes:

1. *Bits 2, 1, and 0 of the ModRM byte select the integer register.*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*

Table 16. MMX™ Technology Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	ModRM byte				
POR mmreg1, mmreg2	0Fh	EBh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
POR mmreg, mem64	0Fh	EBh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSLLD mmreg1, mmreg2	0Fh	F2h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSLLD mmreg, mem64	0Fh	F2h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSLLD mmreg, imm8	0Fh	72h	11-110-xxx	DirectPath	FADD/FMUL	2	
PSLLQ mmreg1, mmreg2	0Fh	F3h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSLLQ mmreg, mem64	0Fh	F3h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSLLQ mmreg, imm8	0Fh	73h	11-110-xxx	DirectPath	FADD/FMUL	2	
PSLLW mmreg1, mmreg2	0Fh	F1h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSLLW mmreg, mem64	0Fh	F1h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSLLW mmreg, imm8	0Fh	71h	11-110-xxx	DirectPath	FADD/FMUL	2	
PSRAD mmreg1, mmreg2	0Fh	E2h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSRAD mmreg, mem64	0Fh	E2h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSRAD mmreg, imm8	0Fh	72h	11-100-xxx	DirectPath	FADD/FMUL	2	
PSRAW mmreg1, mmreg2	0Fh	E1h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSRAW mmreg, mem64	0Fh	E1h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSRAW mmreg, imm8	0Fh	71h	11-100-xxx	DirectPath	FADD/FMUL	2	
PSRLD mmreg1, mmreg2	0Fh	D2h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSRLD mmreg, mem64	0Fh	D2h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSRLD mmreg, imm8	0Fh	72h	11-010-xxx	DirectPath	FADD/FMUL	2	
PSRLQ mmreg1, mmreg2	0Fh	D3h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSRLQ mmreg, mem64	0Fh	D3h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSRLQ mmreg, imm8	0Fh	73h	11-010-xxx	DirectPath	FADD/FMUL	2	
PSRLW mmreg1, mmreg2	0Fh	D1h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSRLW mmreg, mem64	0Fh	D1h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSRLW mmreg, imm8	0Fh	71h	11-010-xxx	DirectPath	FADD/FMUL	2	
PSUBB mmreg1, mmreg2	0Fh	F8h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBB mmreg, mem64	0Fh	F8h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSUBD mmreg1, mmreg2	0Fh	FAh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBD mmreg, mem64	0Fh	FAh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	

Notes:

1. *Bits 2, 1, and 0 of the ModRM byte select the integer register.*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*

Table 16. MMX™ Technology Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	ModRM byte				
PSUBSB mmreg1, mmreg2	0Fh	E8h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBSB mmreg, mem64	0Fh	E8h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSUBSW mmreg1, mmreg2	0Fh	E9h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBSW mmreg, mem64	0Fh	E9h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSUBUSB mmreg1, mmreg2	0Fh	D8h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBUSB mmreg, mem64	0Fh	D8h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSUBUSW mmreg1, mmreg2	0Fh	D9h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBUSW mmreg, mem64	0Fh	D9h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PSUBW mmreg1, mmreg2	0Fh	F9h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PSUBW mmreg, mem64	0Fh	F9h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PUNPCKHBW mmreg1, mmreg2	0Fh	68h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PUNPCKHBW mmreg, mem64	0Fh	68h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PUNPCKHDQ mmreg1, mmreg2	0Fh	6Ah	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PUNPCKHDQ mmreg, mem64	0Fh	6Ah	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PUNPCKHWD mmreg1, mmreg2	0Fh	69h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PUNPCKHWD mmreg, mem64	0Fh	69h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PUNPCKLBW mmreg1, mmreg2	0Fh	60h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PUNPCKLBW mmreg, mem64	0Fh	60h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PUNPCKLDQ mmreg1, mmreg2	0Fh	62h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PUNPCKLDQ mmreg, mem64	0Fh	62h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PUNPCKLWD mmreg1, mmreg2	0Fh	61h	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PUNPCKLWD mmreg, mem64	0Fh	61h	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PXOR mmreg1, mmreg2	0Fh	EFh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PXOR mmreg, mem64	0Fh	EFh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	

Notes:

1. *Bits 2, 1, and 0 of the ModRM byte select the integer register.*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*

C.4 x87 Floating-Point Instructions

Table 17. x87 Floating-Point Instructions

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
F2XM1	D9h		11-110-000	VectorPath	-	65	
FABS	D9h		11-100-001	DirectPath	FMUL	2	
FADD ST, ST(i)	D8h		11-000-xxx	DirectPath	FADD	4	1
FADD [mem32real]	D8h		mm-000-xxx	DirectPath	FADD	6	
FADD ST(i), ST	DCh		11-000-xxx	DirectPath	FADD	4	1
FADD [mem64real]	DCh		mm-000-xxx	DirectPath	FADD	6	
FADDP ST(i), ST	DEh		11-000-xxx	DirectPath	FADD	4	1
FBLD [mem80]	DFh		mm-100-xxx	VectorPath	-	87	
FBSTP [mem80]	DFh		mm-110-xxx	VectorPath	-	172	
FCHS	D9h		11-100-000	DirectPath	FMUL	2	
FCLEX	DBh	E2h	11-100-010	VectorPath	-	~	
FCMOVB ST(0), ST(*i*)	DAh		11-000-xxx	VectorPath	-	15	5
FCMOVBE ST(0), ST(*i*)	DAh		11-010-xxx	VectorPath	-	15	5
FCMOVE ST(0), ST(*i*)	DAh		11-001-xxx	VectorPath	-	15	5
FCMOVNB ST(0), ST(*i*)	DBh		11-000-xxx	VectorPath	-	15	5
FCMOVNBE ST(0), ST(*i*)	DBh		11-010-xxx	VectorPath	-	15	5
FCMOVNE ST(0), ST(*i*)	DBh		11-001-xxx	VectorPath	-	15	5
FCMOVNU ST(0), ST(*i*)	DBh		11-011-xxx	VectorPath	-	15	5
FCMOVU ST(0), ST(*i*)	DAh		11-011-xxx	VectorPath	-	15	5
FCOM ST(i)	D8h		11-010-xxx	DirectPath	FADD	2	1
FCOM [mem32real]	D8h		mm-010-xxx	DirectPath	FADD	4	
FCOM [mem64real]	DCh		mm-010-xxx	DirectPath	FADD	4	
FCOMI ST, ST(i)	DBh		11-110-xxx	VectorPath	FADD	3	3

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

Table 17. x87 Floating-Point Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
FCOMIP ST, ST(i)	DFh		11-110-xxx	VectorPath	FADD	3	3
FCOMP ST(i)	D8h		11-011-xxx	DirectPath	FADD	2	1
FCOMP [mem32real]	D8h		mm-011-xxx	DirectPath	FADD	4	
FCOMP [mem64real]	DCh		mm-011-xxx	DirectPath	FADD	4	
FCOMPP	DEh		11-011-001	DirectPath	FADD	2	
FCOS	D9h		11-111-111	VectorPath	-	92	
FDECSTP	D9h		11-110-110	DirectPath	FADD/FMUL/ FSTORE	2	
FDIV ST, ST(i)	D8h		11-110-xxx	DirectPath	FMUL	16/20 /24	1, 6
FDIV ST(i), ST	DCh		11-111-xxx	DirectPath	FMUL	16/20 /24	1, 6
FDIV [mem32real]	D8h		mm-110-xxx	DirectPath	FMUL	18/22 /26	6
FDIV [mem64real]	DCh		mm-110-xxx	DirectPath	FMUL	18/22 /26	6
FDIVP ST(i), ST	DEh		11-111-xxx	DirectPath	FMUL	16/20 /24	1, 6
FDIVR ST, ST(i)	D8h		11-110-xxx	DirectPath	FMUL	16/20 /24	1, 6
FDIVR ST(i), ST	DCh		11-111-xxx	DirectPath	FMUL	16/20 /24	1, 6
FDIVR [mem32real]	D8h		mm-111-xxx	DirectPath	FMUL	18/22 /26	6
FDIVR [mem64real]	DCh		mm-111-xxx	DirectPath	FMUL	18/22 /26	6
FDIVRP	DEh		11-110-001	DirectPath	FMUL	16/20 /24	6

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. “e” represents the difference between the exponents of the divisor and the dividend. If “s” is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

Table 17. x87 Floating-Point Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
FDIVRP ST(i), ST	DEh		11-110-xxx	DirectPath	FMUL	16/20 /24	1, 6
FFREE ST(i)	DDh		11-000-xxx	DirectPath	FADD/FMUL/ FSTORE	2	1, 2
FIADD [mem32int]	DAh		mm-000-xxx	Double	-	11	
FIADD [mem16int]	DEh		mm-000-xxx	Double	-	11	
FICOM [mem32int]	DAh		mm-010-xxx	Double	-	9	
FICOM [mem16int]	DEh		mm-010-xxx	Double	-	9	
FICOMP [mem32int]	DAh		mm-011-xxx	Double	-	9	
FICOMP [mem16int]	DEh		mm-011-xxx	Double	-	9	
FIDIV [mem32int]	DAh		mm-110-xxx	Double	-	18	
FIDIV [mem16int]	DEh		mm-110-xxx	Double	-	18	
FIDIVR [mem32int]	DAh		mm-111-xxx	Double	-	18	
FIDIVR [mem16int]	DEh		mm-111-xxx	Double	-	18	
FILD [mem16int]	DFh		mm-000-xxx	DirectPath	FSTORE	6	
FILD [mem32int]	DBh		mm-000-xxx	DirectPath	FSTORE	6	
FILD [mem64int]	DFh		mm-101-xxx	DirectPath	FSTORE	6	
FIMUL [mem32int]	DAh		mm-001-xxx	Double	-	11	
FIMUL [mem16int]	DEh		mm-001-xxx	Double	-	11	
FINCSTP	D9h		11-110-111	DirectPath	FADD/FMUL/ FSTORE	2	2
FINIT	DBh		11-100-011	VectorPath	-	~	
FIST [mem16int]	DFh		mm-010-xxx	DirectPath	FSTORE	4	
FIST [mem32int]	DBh		mm-010-xxx	DirectPath	FSTORE	4	
FISTP [mem16int]	DFh		mm-011-xxx	DirectPath	FSTORE	4	
FISTP [mem32int]	DBh		mm-011-xxx	DirectPath	FSTORE	4	

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

Table 17. x87 Floating-Point Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
FISTP [mem64int]	DFh		mm-111-xxx	DirectPath	FSTORE	4	
FISUB [mem32int]	DAh		mm-100-xxx	Double	-	11	
FISUB [mem16int]	DEh		mm-100-xxx	Double	-	11	
FISUBR [mem32int]	DAh		mm-101-xxx	Double	-	11	
FISUBR [mem16int]	DEh		mm-101-xxx	Double	-	11	
FLD ST(i)	D9h		11-000-xxx	DirectPath	FADD/FMUL	2	1
FLD [mem32real]	D9h		mm-000-xxx	DirectPath	FADD/FMUL/ FSTORE	4	
FLD [mem64real]	DDh		mm-000-xxx	DirectPath	FADD/FMUL/ FSTORE	4	
FLD [mem80real]	DBh		mm-101-xxx	VectorPath	-	13	
FLD1	D9h		11-101-000	DirectPath	FSTORE	4	
FLDCW [mem16]	D9h		mm-101-xxx	VectorPath	-	11	
FLDENV [mem14byte]	D9h		mm-100-xxx	VectorPath	-	129	
FLDENV [mem28byte]	D9h		mm-100-xxx	VectorPath	-	129	
FLDL2E	D9h		11-101-010	DirectPath	FSTORE	4	
FLDL2T	D9h		11-101-001	DirectPath	FSTORE	4	
FLDLG2	D9h		11-101-100	DirectPath	FSTORE	4	
FLDLN2	D9h		11-101-101	DirectPath	FSTORE	4	
FLDPI	D9h		11-101-011	DirectPath	FSTORE	4	
FLDZ	D9h		11-101-110	DirectPath	FSTORE	4	
FMUL ST, ST(i)	D8h		11-001-xxx	DirectPath	FMUL	4	1
FMUL ST(i), ST	DCh		11-001-xxx	DirectPath	FMUL	4	1
FMUL [mem32real]	D8h		mm-001-xxx	DirectPath	FMUL	6	
FMUL [mem64real]	DCh		mm-001-xxx	DirectPath	FMUL	6	
FMULP ST(i), ST	DEh		11-001-xxx	DirectPath	FMUL	4	1

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

Table 17. x87 Floating-Point Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
FNCLEX	DBh	E2h		VectorPath		16	
FNINIT	DBh	E3h		VectorPath		89	
FNOP	D9h		11-010-000	DirectPath	FADD/FMUL/ FSTORE	2	2
FPATAN	D9h		11-110-011	VectorPath	-	136	
FPREM	D9h		11-111-000	DirectPath	FMUL	9+e+n	4
FPREM1	D9h		11-110-101	DirectPath	FMUL	9+e+n	4
FPTAN	D9h		11-110-010	VectorPath	-	107	
FRNDINT	D9h		11-111-100	VectorPath	-	10	
FRSTOR [mem94byte]	DDh		mm-100-xxx	VectorPath	-	138	
FRSTOR [mem108byte]	DDh		mm-100-xxx	VectorPath	-	138	
FSAVE [mem94byte]	DDh		mm-110-xxx	VectorPath	-	159	
FSAVE [mem108byte]	DDh		mm-110-xxx	VectorPath	-	159	
FSCALE	D9h		11-111-101	VectorPath	-	9	
FSIN	D9h		11-111-110	VectorPath	-	93	
FSINCOS	D9h		11-111-011	VectorPath	-	104	
FSQRT	D9h		11-111-010	DirectPath	FMUL	35	
FST [mem32real]	D9h		mm-010-xxx	DirectPath	FSTORE	2	
FST [mem64real]	DDh		mm-010-xxx	DirectPath	FSTORE	2	
FST ST(i)	DDh		11-010xxx	DirectPath	FADD/FMUL	2	
FSTCW [mem16]	D9h		mm-111-xxx	VectorPath	-	4	
FSTENV [mem14byte]	D9h		mm-110-xxx	VectorPath	-	89	
FSTENV [mem28byte]	D9h		mm-110-xxx	VectorPath	-	89	
FSTP [mem32real]	D9h		mm-011-xxx	DirectPath	FADD/FMUL	2	
FSTP [mem64real]	DDh		mm-011-xxx	DirectPath	FADD/FMUL	2	

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

Table 17. x87 Floating-Point Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
FSTP [mem80real]	D9h		mm-111-xxx	VectorPath	-	8	
FSTP ST(i)	DDh		11-011-xxx	DirectPath	FADD/FMUL	2	
FSTSW AX	DFh		11-100-000	VectorPath	-	12	
FSTSW [mem16]	DDh		mm-111-xxx	VectorPath	FSTORE	8	3
FSUB [mem32real]	D8h		mm-100-xxx	DirectPath	FADD	6	
FSUB [mem64real]	DCh		mm-100-xxx	DirectPath	FADD	6	
FSUB ST, ST(i)	D8h		11-100-xxx	DirectPath	FADD	4	1
FSUB ST(i), ST	DCh		11-101-xxx	DirectPath	FADD	4	1
FSUBP ST(i), ST	DEh		11-101-xxx	DirectPath	FADD	4	1
FSUBR [mem32real]	D8h		mm-101-xxx	DirectPath	FADD	6	
FSUBR [mem64real]	DCh		mm-101-xxx	DirectPath	FADD	6	
FSUBR ST, ST(i)	D8h		11-100-xxx	DirectPath	FADD	4	1
FSUBR ST(i), ST	DCh		11-101-xxx	DirectPath	FADD	4	1
FSUBRP ST(i), ST	DEh		11-100-xxx	DirectPath	FADD	4	1
FTST	D9h		11-100-100	DirectPath	FADD	2	
FUCOM	DDh		11-100-xxx	DirectPath	FADD	2	
FUCOMI ST, ST(i)	DBh		11-101-xxx	VectorPath	FADD	3	3
FUCOMIP ST, ST(i)	DFh		11-101-xxx	VectorPath	FADD	3	3
FUCOMP	DDh		11-101-xxx	DirectPath	FADD	2	
FUCOMPP	DAh		11-101-001	DirectPath	FADD	2	
FWAIT	9Bh			DirectPath	-	0	
FXAM	D9h		11-100-101	VectorPath	-	2	
FXCH	D9h		11-001-xxx	DirectPath	FADD/FMUL/ FSTORE	2	2
FXRSTOR [mem512byte]	0Fh	AEh	mm-001-xxx	VectorPath	-	68 (108)	

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

Table 17. x87 Floating-Point Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	First byte	Second byte	ModRM byte				
FXSAVE [mem512byte]	0Fh	AEh	mm-000-xxx	VectorPath	-	31 (79)	
FXTRACT	D9h		11-110-100	VectorPath	-	9	
FYL2X	D9h		11-110-001	VectorPath	-	~	
FYL2XP1	D9h		11-111-001	VectorPath	-	113	

Notes:

1. *The last three bits of the ModRM byte select the stack entry ST(i).*
2. *These instructions have an effective latency as shown. However, these instructions generate an internal NOP with a latency of two cycles but no related dependencies. These internal NOPs can be executed at a rate of three per cycle and can use any of the three execution resources.*
3. *This is a VectorPath decoded operation that uses one execution pipe (one ROP).*
4. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).*
5. *The latency provided for this operation is the best-case latency.*
6. *The three latency numbers represent the latency values for precision control settings of single precision, double precision, and extended precision, respectively.*

C.5 3DNow!™ Technology Instructions

Table 18. 3DNow!™ Technology Instructions

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	Prefix byte(s)	imm8	ModRM byte				
FEMMS	0Fh	0Eh		DirectPath	FADD/FMUL/ FSTORE	2	2
PAVGUSB mmreg1, mmreg2	0Fh, 0Fh	BFh	11-xxx-xxx	DirectPath	FADD/FMUL	2	
PAVGUSB mmreg, mem64	0Fh, 0Fh	BFh	mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PF2ID mmreg1, mmreg2	0Fh, 0Fh	1Dh	11-xxx-xxx	DirectPath	FADD	4	
PF2ID mmreg, mem64	0Fh, 0Fh	1Dh	mm-xxx-xxx	DirectPath	FADD	6	
PFACC mmreg1, mmreg2	0Fh, 0Fh	AEh	11-xxx-xxx	DirectPath	FADD	4	
PFACC mmreg, mem64	0Fh, 0Fh	AEh	mm-xxx-xxx	DirectPath	FADD	6	
PFADD mmreg1, mmreg2	0Fh, 0Fh	9Eh	11-xxx-xxx	DirectPath	FADD	4	
PFADD mmreg, mem64	0Fh, 0Fh	9Eh	mm-xxx-xxx	DirectPath	FADD	6	
PFCMPEQ mmreg1, mmreg2	0Fh, 0Fh	B0h	11-xxx-xxx	DirectPath	FADD	2	
PFCMPEQ mmreg, mem64	0Fh, 0Fh	B0h	mm-xxx-xxx	DirectPath	FADD	4	
PFCMPGE mmreg1, mmreg2	0Fh, 0Fh	90h	11-xxx-xxx	DirectPath	FADD	2	
PFCMPGE mmreg, mem64	0Fh, 0Fh	90h	mm-xxx-xxx	DirectPath	FADD	4	
PFCMPGT mmreg1, mmreg2	0Fh, 0Fh	A0h	11-xxx-xxx	DirectPath	FADD	2	
PFCMPGT mmreg, mem64	0Fh, 0Fh	A0h	mm-xxx-xxx	DirectPath	FADD	4	
PFMAX mmreg1, mmreg2	0Fh, 0Fh	A4h	11-xxx-xxx	DirectPath	FADD	2	
PFMAX mmreg, mem64	0Fh, 0Fh	A4h	mm-xxx-xxx	DirectPath	FADD	4	
PFMIN mmreg1, mmreg2	0Fh, 0Fh	94h	11-xxx-xxx	DirectPath	FADD	2	
PFMIN mmreg, mem64	0Fh, 0Fh	94h	mm-xxx-xxx	DirectPath	FADD	4	
PFMUL mmreg1, mmreg2	0Fh, 0Fh	B4h	11-xxx-xxx	DirectPath	FMUL	4	
PFMUL mmreg, mem64	0Fh, 0Fh	B4h	mm-xxx-xxx	DirectPath	FMUL	6	
PFRCP mmreg1, mmreg2	0Fh, 0Fh	96h	11-xxx-xxx	DirectPath	FMUL	3	
PFRCP mmreg, mem64	0Fh, 0Fh	96h	mm-xxx-xxx	DirectPath	FMUL	5	
PFRCPIT1 mmreg1, mmreg2	0Fh, 0Fh	A6h	11-xxx-xxx	DirectPath	FMUL	4	
PFRCPIT1 mmreg, mem64	0Fh, 0Fh	A6h	mm-xxx-xxx	DirectPath	FMUL	6	
PFRCPIT2 mmreg1, mmreg2	0Fh, 0Fh	B6h	11-xxx-xxx	DirectPath	FMUL	4	
PFRCPIT2 mmreg, mem64	0Fh, 0Fh	B6h	mm-xxx-xxx	DirectPath	FMUL	6	
PFRSQIT1 mmreg1, mmreg2	0Fh, 0Fh	A7h	11-xxx-xxx	DirectPath	FMUL	4	

Notes:

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

Table 18. 3DNow!™ Technology Instructions (Continued)

Syntax	Encoding			Decode type	FPU pipe(s)	Latency	Note
	Prefix byte(s)	imm8	ModRM byte				
PFRSQIT1 mmreg, mem64	0Fh, 0Fh	A7h	mm-xxx-xxx	DirectPath	FMUL	6	
PFRSQRT mmreg1, mmreg2	0Fh, 0Fh	97h	11-xxx-xxx	DirectPath	FMUL	3	
PFRSQRT mmreg, mem64	0Fh, 0Fh	97h	mm-xxx-xxx	DirectPath	FMUL	5	
PFSUB mmreg1, mmreg2	0Fh, 0Fh	9Ah	11-xxx-xxx	DirectPath	FADD	4	
PFSUB mmreg, mem64	0Fh, 0Fh	9Ah	mm-xxx-xxx	DirectPath	FADD	6	
PFSUBR mmreg1, mmreg2	0Fh, 0Fh	AAh	11-xxx-xxx	DirectPath	FADD	4	
PFSUBR mmreg, mem64	0Fh, 0Fh	AAh	mm-xxx-xxx	DirectPath	FADD	6	
PI2FD mmreg1, mmreg2	0Fh, 0Fh	0Dh	11-xxx-xxx	DirectPath	FADD	4	
PI2FD mmreg, mem64	0Fh, 0Fh	0Dh	mm-xxx-xxx	DirectPath	FADD	6	
PMULHRW mmreg1, mmreg2	0Fh, 0Fh	B7h	11-xxx-xxx	DirectPath	FMUL	3	
PMULHRW mmreg1, mem64	0Fh, 0Fh	B7h	mm-xxx-xxx	DirectPath	FMUL	5	
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 to be prefetched.*
2. *The byte listed in the column titled 'imm8' is actually the opcode byte.*

C.6 3DNow!™ Technology Extensions

Table 19. 3DNow!™ Technology Extensions

Syntax	Encoding			Decode type	FPU pipe(s)	Latency
	Prefix byte(s)	imm8	ModRM byte			
PF2IW mmreg1, mmreg2	0Fh, 0Fh	1Ch	11-xxx-xxx	DirectPath	FADD	4
PF2IW mmreg, mem64	0Fh, 0Fh	1Ch	mm-xxx-xxx	DirectPath	FADD	6
PFNACC mmreg1, mmreg2	0Fh, 0Fh	8Ah	11-xxx-xxx	DirectPath	FADD	4
PFNACC mmreg, mem64	0Fh, 0Fh	8Ah	mm-xxx-xxx	DirectPath	FADD	6
PFPNACC mmreg1, mmreg2	0Fh, 0Fh	8Eh	11-xxx-xxx	DirectPath	FADD	4
PFPNACC mmreg, mem64	0Fh, 0Fh	8Eh	mm-xxx-xxx	DirectPath	FADD	6
PI2FW mmreg1, mmreg2	0Fh, 0Fh	0Ch	11-xxx-xxx	DirectPath	FADD	4
PI2FW mmreg, mem64	0Fh, 0Fh	0Ch	mm-xxx-xxx	DirectPath	FADD	6
PSWAPD mmreg1, mmreg2	0Fh, 0Fh	BBh	11-xxx-xxx	DirectPath	FADD/FMUL	2
PSWAPD mmreg, mem64	0Fh, 0Fh	BBh	mm-xxx-xxx	DirectPath	FADD/FMUL	4

C.7 SSE Instructions

Table 20. SSE Instructions

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
ADDPS xmmreg1, xmmreg2	0Fh	58h		11-xxx-xxx	Double	FADD	5	1
ADDPS xmmreg, mem128	0Fh	58h		mm-xxx-xxx	Double	FADD	7	1
ADDSS xmmreg1, xmmreg2	F3h	0Fh	58h	11-xxx-xxx	DirectPath	FADD	4	
ADDSS xmmreg, mem128	F3h	0Fh	58h	mm-xxx-xxx	DirectPath	FADD	6	
ANDNPS xmmreg1, xmmreg2	0Fh	55h		11-xxx-xxx	Double	FMUL	3	1
ANDNPS xmmreg, mem128	0Fh	55h		mm-xxx-xxx	Double	FMUL	5	1
ANDPS xmmreg1, xmmreg2	0Fh	54h		11-xxx-xxx	Double	FMUL	3	1
ANDPS xmmreg, mem128	0Fh	54h		mm-xxx-xxx	Double	FMUL	5	1
CMPPS xmmreg1, xmmreg2, imm8	0Fh	C2h		11-xxx-xxx	Double	FADD	3	1
CMPPS xmmreg, mem128, imm8	0Fh	C2h		mm-xxx-xxx	Double	FADD	5	1
CMPSS xmmreg1, xmmreg2, imm8	F3h	0Fh	C2h	11-xxx-xxx	DirectPath	FADD	2	
CMPSS xmmreg, mem32, imm8	F3h	0Fh	C2h	mm-xxx-xxx	DirectPath	FADD	4	
COMISS xmmreg1, xmmreg2	0Fh	2Fh		11-xxx-xxx	VectorPath		4	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
COMISS xmmreg, mem32	0Fh	2Fh		mm-xxx-xxx	VectorPath		6	
CVTPI2PS xmmreg, mmreg	0Fh	2Ah		11-xxx-xxx	DirectPath		4	
CVTPI2PS xmmreg, mem64	0Fh	2Ah		mm-xxx-xxx	DirectPath		6	
CVTPS2PI mmreg, xmmreg	0Fh	2Dh		11-xxx-xxx	DirectPath		4	
CVTPS2PI mmreg, mem128	0Fh	2Dh		mm-xxx-xxx	DirectPath		6	
CVTSI2SS xmmreg, reg32/64	F3h	0Fh	2Ah	11-xxx-xxx	VectorPath		14	
CVTSI2SS xmmreg, mem32/64	F3h	0Fh	2Ah	mm-xxx-xxx	Double		9	
CVTSS2SI reg32, xmmreg	F3h	0Fh	2Dh	11-xxx-xxx	Double		9	
CVTSS2SI reg32, mem32	F3h	0Fh	2Dh	mm-xxx-xxx	VectorPath		10	
CVTTPS2PI mmreg, xmmreg	0Fh	2Ch		11-xxx-xxx	DirectPath		4	
CVTTPS2PI mmreg, mem128	0Fh	2Ch		mm-xxx-xxx	DirectPath		6	
CVTTSS2SI reg32, xmmreg	F3h	0Fh	2Ch	11-xxx-xxx	Double		9	
CVTTSS2SI reg32, mem32	F3h	0Fh	2Ch	mm-xxx-xxx	VectorPath		10	
DIVPS xmmreg1, xmmreg2	0Fh	5Eh		11-xxx-xxx	Double	FMUL	33	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
DIVPS xmmreg, mem128	0Fh	5Eh		mm-xxx-xxx	Double	FMUL	35	
DIVSS xmmreg1, xmmreg2	F3h	0Fh	5Eh	11-xxx-xxx	DirectPath	FMUL	16	
DIVSS xmmreg, mem32	F3h	0Fh	5Eh	mm-xxx-xxx	DirectPath	FMUL	18	
LDMXCSR mem32	0Fh	AEh		mm-010-xxx	VectorPath		13	
MASKMOVQ mmreg1, mmreg2	0Fh	F7h		11-xxx-xxx	VectorPath	FADD/FMUL/ FSTORE	29	
MAXPS xmmreg1, xmmreg2	0Fh	5Fh		11-xxx-xxx	Double	FADD	3	1
MAXPS xmmreg, mem128	0Fh	5Fh		mm-xxx-xxx	Double	FADD	5	1
MAXSS xmmreg1, xmmreg2	F3h	0Fh	5Fh	11-xxx-xxx	DirectPath	FADD	2	
MAXSS xmmreg, mem32	F3h	0Fh	5Fh	mm-xxx-xxx	DirectPath	FADD	4	
MINPS xmmreg1, xmmreg2	0Fh	5Dh		11-xxx-xxx	Double	FADD	3	1
MINPS xmmreg, mem128	0Fh	5Dh		mm-xxx-xxx	Double	FADD	5	1
MINSS xmmreg1, xmmreg2	F3h	0Fh	5Dh	11-xxx-xxx	DirectPath	FADD	2	
MINSS xmmreg, mem32	F3h	0Fh	5Dh	mm-xxx-xxx	DirectPath	FADD	4	
MOVAPS xmmreg1, xmmreg2	0Fh	28h		11-xxx-xxx	Double		2	
MOVAPS xmmreg, mem128	0Fh	28h		mm-xxx-xxx	Double		4	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
MOVAPS xmmreg1, xmmreg2	0Fh	29h		11-xxx-xxx	Double		2	
MOVAPS mem128, xmmreg	0Fh	29h		mm-xxx-xxx	Double		3	1
MOVHLPS xmmreg1, xmmreg2	0Fh	12h		11-xxx-xxx	DirectPath		2	
MOVHPS xmmreg, mem64	0Fh	16h		mm-xxx-xxx	DirectPath		2	
MOVHPS mem64, xmmreg	0Fh	17h		mm-xxx-xxx	DirectPath		2	
MOVLHPS xmmreg1, xmmreg2	0Fh	16h		11-xxx-xxx	DirectPath		2	
MOVLPS xmmreg, mem64	0Fh	12h		mm-xxx-xxx	DirectPath		2	
MOVLPS mem64, xmmreg	0Fh	13h		mm-xxx-xxx	DirectPath		2	
MOVMSKPS reg32, xmmreg	0Fh	50h		11-xxx-xxx	VectorPath		3	
MOVNTPS mem128, xmmreg	0Fh	2Bh		mm-xxx-xxx	Double		3	7
MOVNTQ mem64, mmreg	0Fh	E7h		mm-xxx-xxx	DirectPath	FSTORE	2	7
MOVSS xmmreg1, xmmreg2	F3h	0Fh	10h	11-xxx-xxx	DirectPath		2	
MOVSS xmmreg, mem32	F3h	0Fh	10h	mm-xxx-xxx	Double		3	
MOVSS xmmreg1, xmmreg2	F3h	0Fh	11h	11-xxx-xxx	DirectPath		2	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
MOVSS mem32, xmmreg	F3h	0Fh	11h	mm-xxx-xxx	DirectPath		2	
MOVUPS xmmreg1, xmmreg2	0Fh	10h		11-xxx-xxx	Double		2	
MOVUPS xmmreg, mem128	0Fh	10h		mm-xxx-xxx	VectorPath		7	
MOVUPS xmmreg1, xmmreg2	0Fh	11h		11-xxx-xxx	Double		2	
MOVUPS mem128, xmmreg	0Fh	11h		mm-xxx-xxx	VectorPath		4	
MULPS xmmreg1, xmmreg2	0Fh	59h		11-xxx-xxx	Double	FMUL	5	1
MULPS xmmreg, mem128	0Fh	59h		mm-xxx-xxx	Double	FMUL	7	1
MULSS xmmreg1, xmmreg2	F3h	0Fh	59h	11-xxx-xxx	DirectPath	FMUL	4	
MULSS xmmreg, mem32	F3h	0Fh	59h	mm-xxx-xxx	DirectPath	FMUL	6	
ORPS xmmreg1, xmmreg2	0Fh	56h		11-xxx-xxx	Double	FMUL	3	1
ORPS xmmreg, mem128	0Fh	56h		mm-xxx-xxx	Double	FMUL	5	1
PAVGB mmreg1, mmreg2	0Fh	E0h		11-xxx-xxx	DirectPath	FADD/FMUL	2	
PAVGB mmreg, mem64	0Fh	E0h		mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PAVGW mmreg1, mmreg2	0Fh	E3h		11-xxx-xxx	DirectPath	FADD/FMUL	2	
PAVGW mmreg, mem64	0Fh	E3h		mm-xxx-xxx	DirectPath	FADD/FMUL	4	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
PEXTRW reg32/64, mmreg, imm8	0Fh	C5h			Double	-	4	4
PINSRW mmreg, reg32/64, imm8	0Fh	C4h			Double	-	9	4
PINSRW mmreg, mem16, imm8	0Fh	C4h			DirectPath	-	4	4
PMAXSW mmreg1, mmreg2	0Fh	EEh		11-xxx-xxx	DirectPath	FADD/FMUL	2	
PMAXSW mmreg, mem64	0Fh	EEh		mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PMAXUB mmreg1, mmreg2	0Fh	DEh		11-xxx-xxx	DirectPath	FADD/FMUL	2	
PMAXUB mmreg, mem64	0Fh	DEh		mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PMINSW mmreg1, mmreg2	0Fh	EAh		11-xxx-xxx	DirectPath	FADD/FMUL	2	
PMINSW mmreg, mem64	0Fh	EAh		mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PMINUB mmreg1, mmreg2	0Fh	DAh		11-xxx-xxx	DirectPath	FADD/FMUL	2	
PMINUB mmreg, mem64	0Fh	DAh		mm-xxx-xxx	DirectPath	FADD/FMUL	4	
PMOVMSKB reg32/64, mmreg	0Fh	D7h			VectorPath	-	3	4
PMULHUW mmreg1, mmreg2	0Fh	E4h		11-xxx-xxx	DirectPath	FMUL	3	
PMULHUW mmreg, mem64	0Fh	E4h		mm-xxx-xxx	DirectPath	FMUL	5	
PREFETCHNTA mem8	0Fh	18h		mm-000-xxx	DirectPath	~	~	5

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
PREFETCHT0 mem8	0Fh	18h		mm-001-xxx	DirectPath	~	~	5
PREFETCHT1 mem8	0Fh	18h		mm-010-xxx	DirectPath	~	~	5
PREFETCHT2 mem8	0Fh	18h		mm-011-xxx	DirectPath	~	~	5
PSADBW mmreg1, mmreg2	0Fh	F6h		11-xxx-xxx	DirectPath	FADD	3	
PSADBW mmreg, mem64	0Fh	F6h		mm-xxx-xxx	DirectPath	FADD	5	
PSHUFW mmreg1, mmreg2, imm8	0Fh	70h			DirectPath	FADD/FMUL	2	
PSHUFW mmreg, mem64, imm8	0Fh	70h			DirectPath	FADD/FMUL	4	
RCPPS xmmreg1, xmmreg2	0Fh	53h		11-xxx-xxx	Double	FMUL	4	1
RCPPS xmmreg, mem128	0Fh	53h		mm-xxx-xxx	Double	FMUL	6	1
RCPSS xmmreg1, xmmreg2	F3h	0Fh	53h	11-xxx-xxx	DirectPath	FMUL	3	
RCPSS xmmreg, mem32	F3h	0Fh	53h	mm-xxx-xxx	DirectPath	FMUL	5	
RSQRTPS xmmreg1, xmmreg2	0Fh	52h		11-xxx-xxx	Double	FMUL	4	1
RSQRTPS xmmreg, mem128	0Fh	52h		mm-xxx-xxx	Double	FMUL	6	1
RSQRTSS xmmreg1, xmmreg2	F3h	0Fh	52h	11-xxx-xxx	DirectPath	FMUL	3	
RSQRTSS xmmreg, mem32	F3h	0Fh	52h	mm-xxx-xxx	DirectPath	FMUL	5	
SFENCE	0Fh	AEh		11-111-000	VectorPath		2/8	6

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
SHUFPS xmmreg1, xmmreg2, imm8	0Fh	C6h		11-xxx-xxx	VectorPath	FMUL	4	1
SHUFPS xmmreg, mem128, imm8	0Fh	C6h		mm-xxx-xxx	VectorPath	FMUL	6	2
SQRTPS xmmreg1, xmmreg2	0Fh	51h		11-xxx-xxx	Double	FMUL	39	
SQRTPS xmmreg, mem128	0Fh	51h		mm-xxx-xxx	Double	FMUL	41	
SQRTSS xmmreg1, xmmreg2	F3h	0Fh	51h	11-xxx-xxx	DirectPath	FMUL	19	
SQRTSS xmmreg, mem32	F3h	0Fh	51h	mm-xxx-xxx	DirectPath	FMUL	21	
STMXCSR mem32	0Fh	AEh		mm-011-xxx	VectorPath		11	
SUBPS xmmreg1, xmmreg2	0Fh	5Ch		11-xxx-xxx	Double	FADD	5	1
SUBPS xmmreg, mem128	0Fh	5Ch		mm-xxx-xxx	Double	FADD	7	1
SUBSS xmmreg1, xmmreg2	F3h	0Fh	5Ch	11-xxx-xxx	DirectPath	FADD	4	
SUBSS xmmreg, mem32	F3h	0Fh	5Ch	mm-xxx-xxx	DirectPath	FADD	6	
UCOMISS xmmreg1, xmmreg2	0Fh	2Eh		11-xxx-xxx	VectorPath		4	
UCOMISS xmmreg, mem32	0Fh	2Eh		mm-xxx-xxx	VectorPath		6	
UNPCKHPS xmmreg1, xmmreg2	0Fh	15h		11-xxx-xxx	Double	FMUL	3	1
UNPCKHPS xmmreg, mem128	0Fh	15h		mm-xxx-xxx	Double	FMUL	5	1

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 20. SSE Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Note
	Prefix byte	First byte	2nd byte	ModRM byte				
UNPCKLPS xmmreg1, xmmreg2	0Fh	14h		11-xxx-xxx	Double	FMUL	3	3
UNPCKLPS xmmreg, mem128	0Fh	14h		mm-xxx-xxx	Double	FMUL	5	3
XORPS xmmreg1, xmmreg2	0Fh	57h		11-xxx-xxx	Double	FMUL	3	1
XORPS xmmreg, mem128	0Fh	57h		mm-xxx-xxx	Double	FMUL	5	1

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *The second latency value indicates when the low half of the result becomes available.*
3. *The high half of the result is available one cycle earlier than listed.*
4. *The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.*
5. *For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line to be prefetched.*
6. *The 8-clock latency is only visible to younger stores that need to do an external write. The 2-clock latency is visible to the other stores and instructions.*
7. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

C.8 SSE2 Instructions

Table 21. SSE2 Instructions

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
ADDPD xmmreg1, xmmreg2	66h	0Fh	58h		Double	FADD	5	1/2	
ADDPD xmmreg, mem128	66h	0Fh	58h		Double	FADD	7	1/2	
ADDSD xmmreg1, xmmreg2	F2h	0Fh	58h		DirectPath	FADD	4	1/1	
ADDSD xmmreg, mem64	F2h	0Fh	58h		DirectPath	FADD	6	1/1	
ANDNPD xmmreg1, xmmreg2	66h	0Fh	55h		Double	FMUL	3	1/2	
ANDNPD xmmreg, mem128	66h	0Fh	55h		Double	FMUL	5	1/2	
ANDPD xmmreg1, xmmreg2	66h	0Fh	54h		Double	FMUL	3	1/2	
ANDPD xmmreg, mem128	66h	0Fh	54h		Double	FMUL	5	1/2	
CMPPD xmmreg1, xmmreg2, imm8	66h	0Fh	C2h		Double	FADD	3	1/2	
CMPPD xmmreg, mem128, imm8	66h	0Fh	C2h		Double	FADD	5	1/2	
CMPSD xmmreg1, xmmreg2, imm8	F2h	0Fh	C2h		DirectPath	FADD	2	1/1	
CMPSD xmmreg, mem64, imm8	F2h	0Fh	C2h		DirectPath	FADD	4	1/1	
COMISD xmmreg1, xmmreg2	66h	0Fh	2Fh		VectorPath	FADD	4	1	
COMISD xmmreg, mem64	66h	0Fh	2Fh		VectorPath	FADD	5	1	
CVTDQ2PD xmmreg1, xmmreg2	F3h	0Fh	E6h		Double	FSTORE	5	1/2	
CVTDQ2PD xmmreg, mem64	F3h	0Fh	E6h		Double	FSTORE	7	1/2	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
CVTDQ2PS xmmreg1, xmmreg2	0Fh	5Bh			Double	FSTORE	5	1/2	
CVTDQ2PS xmmreg, mem128	0Fh	5Bh			Double	FSTORE	7	1/2	
CVTPD2DQ xmmreg1, xmmreg2	F2h	0Fh	E6h		VectorPath	~	8		
CVTPD2DQ xmmreg, mem128	F2h	0Fh	E6h		VectorPath	~	10		
CVTPD2PI mmreg, xmmreg	66h	0Fh	2Dh		VectorPath	~	8	1/2	
CVTPD2PI mmreg, mem128	66h	0Fh	2Dh		VectorPath	~	10	1/2	
CVTPD2PS xmmreg1, xmmreg2	66h	0Fh	5Ah		VectorPath	~	8		
CVTPD2PS xmmreg, mem128	66h	0Fh	5Ah		VectorPath	~	10		
CVTPI2PD xmmreg, mmreg	66H	0FH	2Ah		Double	FSTORE	5	1/2	
CVTPI2PD xmmreg, mem64	66H	0FH	2Ah		Double	FSTORE	7	1/2	
CVTPS2DQ xmmreg1, xmmreg2	66h	0Fh	5Bh		Double	FSTORE	5	1/2	
CVTPS2DQ xmmreg, mem128	66h	0Fh	5Bh		Double	FSTORE	7	1/2	
CVTPS2PD xmmreg1, xmmreg2	0Fh	5Ah			Double	~	3	1/2	
CVTPS2PD xmmreg, mem64	0Fh	5Ah			Double	~	5	1/2	
CVTSD2SI reg32/64, xmmreg	F2h	0Fh	2Dh		Double	FSTORE	9	1/1	
CVTSD2SI reg32/64, mem64	F2h	0Fh	2Dh		VectorPath	FADD/ FMUL/ FSTORE	10	1/1	
CVTSD2SS xmmreg1, xmmreg2	F2h	0Fh	5Ah		VectorPath	FSTORE	12		

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
CVTSD2SS xmmreg, mem64	F2h	0Fh	5Ah		Double	FSTORE	9		
CVTSI2SD xmmreg, reg32/64	F2h	0Fh	2Ah		Double	FSTORE	11	1/1	
CVTSI2SD xmmreg, mem32/64	F2h	0Fh	2Ah		DirectPath	FSTORE	6	1/1	
CVTSS2SD xmmreg1, xmmreg2	F3h	0Fh	5Ah		DirectPath	FSTORE	2	1/1	
CVTSS2SD xmmreg, mem32	F3h	0Fh	5Ah		DirectPath	FSTORE	4	1/1	
CVTSS2SI reg32/64, xmmreg	F3h	0Fh	2Dh		Double	FSTORE	9		
CVTSS2SI reg32/64, mem32	F3h	0Fh	2Dh		VectorPath	~	10		
CVTTPD2DQ xmmreg1, xmmreg2	66h	0Fh	E6h		VectorPath	~	8		
CVTTPD2DQ xmmreg, mem128	66h	0Fh	E6h		VectorPath	~	10		
CVTTPD2PI mmreg, xmmreg	66h	0Fh	2Ch		VectorPath	~	8	1/2	
CVTTPD2PI mmreg, mem128	66h	0Fh	2Ch		VectorPath	~	10	1/2	
CVTTPS2DQ xmmreg1, xmmreg2	F3h	0Fh	5Bh		Double	FSTORE	5	1/2	
CVTTPS2DQ xmmreg, mem128	F3h	0Fh	5Bh		Double	FSTORE	7	1/2	
CVTTSD2SI reg32/64, xmmreg	F2h	0Fh	2Ch		Double	FSTORE	9	1/1	
CVTTSD2SI reg32/64, mem64	F2h	0Fh	2Ch		VectorPath	FADD/ FMUL/ FSTORE	10	1/1	
CVTTSS2SI reg32/64, xmmreg	F3h	0Fh	2Ch		Double	FSTORE	9		
CVTTSS2SI reg32/64, mem32	F3h	0Fh	2Ch		VectorPath	~	10		

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
DIVPD xmmreg1, xmmreg2	66h	0Fh	5Eh		Double	FMUL	37	1/34	
DIVPD xmmreg, mem128	66h	0Fh	5Eh		Double	FMUL	39	1/34	
DIVSD xmmreg1, xmmreg2	F2h	0Fh	5Eh		DirectPath	FMUL	20	1/17	
DIVSD xmmreg, mem64	F2h	0Fh	5Eh		DirectPath	FMUL	22	1/17	
MASKMOVDQU xmmreg1, xmmreg2	66h	0Fh	F7h		VectorPath	~	43		
MAXPD xmmreg1, xmmreg2	66h	0Fh	5Fh		Double	FADD	3	1/2	
MAXPD xmmreg, mem128	66h	0Fh	5Fh		Double	FADD	5	1/2	
MAXSD xmmreg1, xmmreg2	F2h	0Fh	5Fh		DirectPath	FADD	2	1/1	
MAXSD xmmreg, mem64	F2h	0Fh	5Fh		DirectPath	FADD	4	1/1	
MINPD xmmreg1, xmmreg2	66h	0Fh	5Dh		Double	FADD	3	1/2	
MINPD xmmreg, mem128	66h	0Fh	5Dh		Double	FADD	5	1/2	
MINSD xmmreg1, xmmreg2	F2h	0Fh	5Dh		DirectPath	FADD	2	1/1	
MINSD xmmreg, mem64	F2h	0Fh	5Dh		DirectPath	FADD	4	1/1	
MOVAPD xmmreg1, xmmreg2	66h	0Fh	28h		Double	FADD/ FMUL	2		
MOVAPD xmmreg, mem128	66h	0Fh	28h		Double	FADD/ FMUL/ FSTORE	2		
MOVAPD xmmreg1, xmmreg2	66h	0Fh	29h		Double	FADD/ FMUL	2		
MOVAPD mem128, xmmreg	66h	0Fh	29h		Double	FSTORE	3		

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
MOVD xmmreg, reg32	66h	0Fh	6Eh		VectorPath	~	9		
MOVD xmmreg, mem32	66h	0Fh	6Eh		Double	FADD/ FMUL/ FSTORE	4		
MOVD reg32, xmmreg	66h	0Fh	7Eh		Double	FSTORE	4		
MOVD mem32, xmmreg	66h	0Fh	7Eh		DirectPath	FSTORE	2		
MOVD xmmreg, reg64	66h	0Fh	6Eh		VectorPath	~	9		
MOVD xmmreg, mem64	66h	0Fh	6Eh		Double	FADD/ FMUL/ FSTORE	4		
MOVD reg64, xmmreg	66h	0Fh	7Eh		Double	FSTORE	4		
MOVD mem64, xmmreg	66h	0Fh	7Eh		DirectPath	FSTORE	2		
MOVDQ2Q mmreg, xmmreg	F2h	0Fh	D6h		DirectPath	FADD/ FMUL	2		
MOVDQA xmmreg1, xmmreg2	66h	0Fh	6Fh		Double	FADD/ FMUL	2		
MOVDQA xmmreg, mem128	66h	0Fh	6Fh		Double	FADD/ FMUL/ FSTORE	2		
MOVDQA xmmreg1, xmmreg2	66h	0Fh	7Fh		Double	FADD/ FMUL	2		
MOVDQA mem128, xmmreg	66h	0Fh	7Fh		Double	FSTORE	3		
MOVDQU xmmreg1, xmmreg2	F3h	0Fh	6Fh		Double	FADD/ FMUL	2		
MOVDQU xmmreg, mem128	F3h	0Fh	6Fh		VectorPath	~	7		
MOVDQU xmmreg1, xmmreg2	F3h	0Fh	7Fh		Double	FADD/ FMUL	2		
MOVDQU mem128, xmmreg	F3h	0Fh	7Fh		VectorPath	FSTORE	4		
MOVHPD xmmreg, mem64	66h	0Fh	16h		DirectPath	FADD/ FMUL/ FSTORE	4		

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
MOVHPD mem64, xmmreg	66h	0Fh	16h		DirectPath	FSTORE	2		
MOVLPD xmmreg, mem64	66h	0Fh	12h		DirectPath	FADD/ FMUL/ FSTORE	4		
MOVLPD mem64, xmmreg	66h	0Fh	13h		DirectPath	FSTORE	2		
MOVMSKPD reg32/64, xmmreg	66h	0Fh	50h		VectorPath	FADD	3	1/1	
MOVNTDQ mem128, xmmreg	66h	0Fh	E7h		Double	FSTORE	3		2
MOVNTI mem32/64, reg32/64		0Fh	C3h	mm-xxx-xxx	DirectPath	FSTORE	~		
MOVNTPD mem128, xmmreg	66h	0Fh	2Bh		Double	FSTORE	3		2
MOVQ xmmreg1, xmmreg2	F3h	0Fh	7Eh		Double	FADD/ FMUL	2		
MOVQ xmmreg, mem64	F3h	0Fh	7Eh		Double	FADD/ FMUL/ FSTORE	4		
MOVQ xmmreg1, xmmreg2	66h	0Fh	D6h		Double	FADD/ FMUL	2		
MOVQ mem64, xmmreg	66h	0Fh	D6h		DirectPath	FSTORE	4		
MOVQ2DQ xmmreg, mmreg	F3h	0Fh	D6h		Double	FADD/ FMUL	2		
MOVSD xmmreg1, xmmreg2	F2h	0Fh	10h		DirectPath	FADD/ FMUL	2		
MOVSD xmmreg, mem64	F2h	0Fh	10h		Double	FADD/ FMUL/ FSTORE	3		1
MOVSD xmmreg1, xmmreg2	F2h	0Fh	11h		DirectPath	FADD/ FMUL	2		
MOVSD mem64, xmmreg	F2h	0Fh	11h		DirectPath	FSTORE	2		

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
MOVUPD xmmreg1, xmmreg2	66h	0Fh	10h		Double	FADD/ FMUL	2		
MOVUPD xmmreg, mem128	66h	0Fh	10h		VectorPath	FADD/ FMUL/ FSTORE	7		
MOVUPD xmmreg1, xmmreg2	66h	0Fh	11h		Double	FADD/ FMUL	2		
MOVUPD mem128, xmmreg	66h	0Fh	11h		VectorPath	FSTORE	4		
MULPD xmmreg1, xmmreg2	66h	0Fh	59h		Double	FMUL	5	1/2	
MULPD xmmreg, mem128	66h	0Fh	59h		Double	FMUL	7	1/2	
MULSD xmmreg1, xmmreg2	F2h	0Fh	59h		DirectPath	FMUL	4	1/1	
MULSD xmmreg, mem64	F2h	0Fh	59h		DirectPath	FMUL	6	1/1	
ORPD xmmreg1, xmmreg2	66h	0Fh	56h		Double	FMUL	3	1/2	
ORPD xmmreg, mem128	66h	0Fh	56h		Double	FMUL	5	1/2	
PACKSSDW xmmreg1, xmmreg2	66h	0Fh	6Bh		VectorPath	~	4		
PACKSSDW xmmreg, mem128	66h	0Fh	6Bh		VectorPath	~	6		
PACKSSWB xmmreg1, xmmreg2	66h	0Fh	63h		VectorPath	~	4		
PACKSSWB xmmreg, mem128	66h	0Fh	63h		VectorPath	~	6		
PACKUSWB xmmreg1, xmmreg2	66h	0Fh	67h		VectorPath	~	4		
PACKUSWB xmmreg, mem128	66h	0Fh	67h		VectorPath	~	6		
PADDB xmmreg1, xmmreg2	66h	0Fh	FCh		Double	FADD/ FMUL	2	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PADDB xmmreg, mem128	66h	0Fh	FCh		Double	FADD/ FMUL	4	1/1	
PADDD xmmreg1, xmmreg2	66h	0Fh	FEh		Double	FADD/ FMUL	2	1/1	
PADDD xmmreg, mem128	66h	0Fh	FEh		Double	FADD/ FMUL	4	1/1	
PADDQ mmreg1, mmreg2	0Fh	D4h			DirectPath	FADD/ FMUL	2	1/1	
PADDQ mmreg, mem64	0Fh	D4h			DirectPath	FADD/ FMUL	4	1/1	
PADDQ xmmreg1, xmmreg2	66h	0Fh	D4h		Double	FADD/ FMUL	2	1/1	
PADDQ xmmreg, mem128	66h	0Fh	D4h		Double	FADD/ FMUL	4	1/1	
PADDSB xmmreg1, xmmreg2	66h	0Fh	ECh		Double	FADD/ FMUL	2	1/1	
PADDSB xmmreg, mem128	66h	0Fh	ECh		Double	FADD/ FMUL	4	1/1	
PADDSW xmmreg1, xmmreg2	66h	0Fh	EDh		Double	FADD/ FMUL	2	1/1	
PADDSW xmmreg, mem128	66h	0Fh	EDh		Double	FADD/ FMUL	4	1/1	
PADDUSB xmmreg1, xmmreg2	66h	0Fh	DCh		Double	FADD/ FMUL	2	1/1	
PADDUSB xmmreg, mem128	66h	0Fh	DCh		Double	FADD/ FMUL	4	1/1	
PADDUSW xmmreg1, xmmreg2	66h	0Fh	DDh		Double	FADD/ FMUL	2	1/1	
PADDUSW xmmreg, mem128	66h	0Fh	DDh		Double	FADD/ FMUL	4	1/1	
PADDW xmmreg1, xmmreg2	66h	0Fh	FDh		Double	FADD/ FMUL	2	1/1	
PADDW xmmreg, mem128	66h	0Fh	FDh		Double	FADD/ FMUL	4	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PAND xmmreg1, xmmreg2	66h	0Fh	DBh		Double	FADD/ FMUL	2	1/1	
PAND xmmreg, mem128	66h	0Fh	DBh		Double	FADD/ FMUL	4	1/1	
PANDN xmmreg1, xmmreg2	66h	0Fh	DFh		Double	FADD/ FMUL	2	1/1	
PANDN xmmreg, mem128	66h	0Fh	DFh		Double	FADD/ FMUL	4	1/1	
PAVGB xmmreg1, xmmreg2	66h	0Fh	E0h		Double	FADD/ FMUL	2	1/1	
PAVGB xmmreg, mem128	66h	0Fh	E0h		Double	FADD/ FMUL	4	1/1	
PAVGW xmmreg1, xmmreg2	66h	0Fh	E3h		Double	FADD/ FMUL	2	1/1	
PAVGW xmmreg, mem128	66h	0Fh	E3h		Double	FADD/ FMUL	4	1/1	
PCMPEQB xmmreg1, xmmreg2	66h	0Fh	74h		Double	FADD/ FMUL	2	1/1	
PCMPEQB xmmreg, mem128	66h	0Fh	74h		Double	FADD/ FMUL	4	1/1	
PCMPEQD xmmreg1, xmmreg2	66h	0Fh	76h		Double	FADD/ FMUL	2	1/1	
PCMPEQD xmmreg, mem128	66h	0Fh	76h		Double	FADD/ FMUL	4	1/1	
PCMPEQW xmmreg1, xmmreg2	66h	0Fh	75h		Double	FADD/ FMUL	2	1/1	
PCMPEQW xmmreg, mem128	66h	0Fh	75h		Double	FADD/ FMUL	4	1/1	
PCMPGTB xmmreg1, xmmreg2	66h	0Fh	64h		Double	FADD/ FMUL	2	1/1	
PCMPGTB xmmreg, mem128	66h	0Fh	64h		Double	FADD/ FMUL	4	1/1	
PCMPGTD xmmreg1, xmmreg2	66h	0Fh	66h		Double	FADD/ FMUL	2	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PCMPGTD xmmreg, mem128	66h	0Fh	66h		Double	FADD/ FMUL	4	1/1	
PCMPGTW xmmreg1, xmmreg2	66h	0Fh	65h		Double	FADD/ FMUL	2	1/1	
PCMPGTW xmmreg, mem128	66h	0Fh	65h		Double	FADD/ FMUL	4	1/1	
PEXTRW reg32/64, xmmreg, imm8	66h	0Fh	C5h		Double	FSTORE	4	1/1	
PINSRW xmmreg, reg32/64, imm8	66h	0Fh	C4h		VectorPath	FADD/ FMUL	10	1/1	
PINSRW xmmreg, mem128, imm8	66h	0Fh	C4h		Double	FADD/ FMUL	4	1/1	
PMADDWD xmmreg1, xmmreg2	66h	0Fh	F5h		Double	FMUL	4	1/2	
PMADDWD xmmreg, mem128	66h	0Fh	F5h		Double	FMUL	6	1/2	
PMAXSW xmmreg1, xmmreg2	66h	0Fh	EEh		Double	FADD/ FMUL	2	1/1	
PMAXSW xmmreg, mem128	66h	0Fh	EEh		Double	FADD/ FMUL	4	1/1	
PMAXUB xmmreg1, xmmreg2	66h	0Fh	DEh		Double	FADD/ FMUL	2	1/1	
PMAXUB xmmreg, mem128	66h	0Fh	DEh		Double	FADD/ FMUL	4	1/1	
PMINSW xmmreg1, xmmreg2	66h	0Fh	EAh		Double	FADD/ FMUL	2	1/1	
PMINSW xmmreg, mem128	66h	0Fh	EAh		Double	FADD/ FMUL	4	1/1	
PMINUB xmmreg1, xmmreg2	66h	0Fh	DAh		Double	FADD/ FMUL	2	1/1	
PMINUB xmmreg, mem128	66h	0Fh	DAh		Double	FADD/ FMUL	4	1/1	
PMOVMSKB reg32/64, xmmreg	66h	0Fh	D7h		VectorPath	FADD/ FMUL	3	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PMULHUW xmmreg1, xmmreg2	66h	0Fh	E4h		Double	FMUL	4	1/2	
PMULHUW xmmreg, mem128	66h	0Fh	E4h		Double	FMUL	6	1/2	
PMULHW xmmreg1, xmmreg2	66h	0Fh	E5h		Double	FMUL	4	1/2	
PMULHW xmmreg, mem128	66h	0Fh	E5h		Double	FMUL	6	1/2	
PMULLW xmmreg1, xmmreg2	66h	0Fh	D5h		Double	FMUL	4	1/2	
PMULLW xmmreg, mem128	66h	0Fh	D5h		Double	FMUL	6	1/2	
PMULUDQ mmreg1, mmreg2	0Fh	F4h			DirectPath	FMUL	3	1/2	
PMULUDQ mmreg, mem64	0Fh	F4h			DirectPath	FMUL	5	1/2	
PMULUDQ xmmreg1, xmmreg2	66h	0Fh	F4h		Double	FMUL	4	1/2	
PMULUDQ xmmreg, mem128	66h	0Fh	F4h		Double	FMUL	6	1/2	
POR xmmreg1, xmmreg2	66h	0Fh	EBh		Double	FADD/ FMUL	2	1/1	
POR xmmreg, mem128	66h	0Fh	EBh		Double	FADD/ FMUL	4	1/1	
PSADBW xmmreg1, xmmreg2	66h	0Fh	F6h		Double	FADD	4	1/2	
PSADBW xmmreg, mem128	66h	0Fh	F6h		Double	FADD	6	1/2	
PSHUFD xmmreg1, xmmreg2, imm8	66h	0Fh	70h		VectorPath	~	4		
PSHUFD xmmreg, mem128, imm8	66h	0Fh	70h		VectorPath	~	6		
PSHUFHW xmmreg1, xmmreg2, imm8	F3h	0Fh	70h		Double	FADD/ FMUL	2	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PSHUFHW xmmreg, mem128, imm8	F3h	0Fh	70h		Double	FADD/ FMUL	4	1/1	
PSHUFLW xmmreg1, xmmreg2, imm8	F2h	0Fh	70h		Double	FADD/ FMUL	2	1/1	
PSHUFLW xmmreg, mem128, imm8	F2h	0Fh	70h		Double	FADD/ FMUL	4	1/1	
PSLLD xmmreg1, xmmreg2	66h	0Fh	F2h		Double	FADD/ FMUL	2	1/1	
PSLLD xmmreg, mem128	66h	0Fh	F2h		Double	FADD/ FMUL	4	1/1	
PSLLD xmmreg, imm8	66h	0Fh	72h		Double	FADD/ FMUL	2	1/1	
PSLLDQ xmmreg, imm8	66h	0Fh	73h	11-111-xxx	Double	FADD/ FMUL	2	1/1	
PSLLQ xmmreg1, xmmreg2	66h	0Fh	F3h		Double	FADD/ FMUL	2	1/1	
PSLLQ xmmreg, mem128	66h	0Fh	F3h		Double	FADD/ FMUL	4	1/1	
PSLLQ xmmreg, imm8	66h	0Fh	73h	11-110-xxx	Double	FADD/ FMUL	2	1/1	
PSLLW xmmreg1, xmmreg2	66h	0Fh	F1h		Double	FADD/ FMUL	2	1/1	
PSLLW xmmreg, mem128	66h	0Fh	F1h		Double	FADD/ FMUL	4	1/1	
PSLLW xmmreg, imm8	66h	0Fh	71h	11-110-xxx	Double	FADD/ FMUL	2	1/1	
PSRAD xmmreg1, xmmreg2	66h	0Fh	E2h		Double	FADD/ FMUL	2	1/1	
PSRAD xmmreg, mem128	66h	0Fh	E2h		Double	FADD/ FMUL	4	1/1	
PSRAD xmmreg, imm8	66h	0Fh	72h	11-100-xxx	Double	FADD/ FMUL	2	1/1	
PSRAW xmmreg1, xmmreg2	66h	0Fh	E1h		Double	FADD/ FMUL	2	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PSRAW xmmreg, mem128	66h	0Fh	E1h		Double	FADD/ FMUL	4	1/1	
PSRAW xmmreg, imm8	66h	0Fh	71h	11-100-xxx	Double	FADD/ FMUL	2	1/1	
PSRLD xmmreg1, xmmreg2	66h	0Fh	D2h		Double	FADD/ FMUL	2	1/1	
PSRLD xmmreg, mem128	66h	0Fh	D2h		Double	FADD/ FMUL	4	1/1	
PSRLD xmmreg, imm8	66h	0Fh	72h	11-010-xxx	Double	FADD/ FMUL	2	1/1	
PSRLDQ xmmreg, imm8	66h	0Fh	73h	11-011-xxx	Double	FADD/ FMUL	2	1/1	
PSRLQ xmmreg1, xmmreg2	66h	0Fh	D3h		Double	FADD/ FMUL	2	1/1	
PSRLQ xmmreg, mem128	66h	0Fh	D3h		Double	FADD/ FMUL	4	1/1	
PSRLQ xmmreg, imm8	66h	0Fh	73h	11-010-xxx	Double	FADD/ FMUL	2	1/1	
PSRLW xmmreg1, xmmreg2	66h	0Fh	D1h		Double	FADD/ FMUL	2	1/1	
PSRLW xmmreg, mem128	66h	0Fh	D1h		Double	FADD/ FMUL	4	1/1	
PSRLW xmmreg, imm8	66h	0Fh	71h	11-010-xxx	Double	FADD/ FMUL	2	1/1	
PSUBB xmmreg1, xmmreg2	66h	0Fh	F8h		Double	FADD/ FMUL	2	1/1	
PSUBB xmmreg, mem128	66h	0Fh	F8h		Double	FADD/ FMUL	4	1/1	
PSUBD xmmreg1, xmmreg2	66h	0Fh	FAh		Double	FADD/ FMUL	2	1/1	
PSUBD xmmreg, mem128	66h	0Fh	FAh		Double	FADD/ FMUL	4	1/1	
PSUBQ mmreg1, mmreg2	0Fh	FBh			DirectPath	FADD/ FMUL	2	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PSUBQ mmreg, mem64	0Fh	FBh			DirectPath	FADD/ FMUL	5	1/1	
PSUBQ xmmreg1, xmmreg2	66h	0Fh	FBh		Double	FADD/ FMUL	2	1/1	
PSUBQ xmmreg, mem128	66h	0Fh	FBh		Double	FADD/ FMUL	4	1/1	
PSUBSB xmmreg1, xmmreg2	66h	0Fh	E8h		Double	FADD/ FMUL	2	1/1	
PSUBSB xmmreg, mem128	66h	0Fh	E8h		Double	FADD/ FMUL	4	1/1	
PSUBSW xmmreg1, xmmreg2	66h	0Fh	E9h		Double	FADD/ FMUL	2	1/1	
PSUBSW xmmreg, mem128	66h	0Fh	E9h		Double	FADD/ FMUL	4	1/1	
PSUBUSB xmmreg1, xmmreg2	66h	0Fh	D8h		Double	FADD/ FMUL	2	1/1	
PSUBUSB xmmreg, mem128	66h	0Fh	D8h		Double	FADD/ FMUL	4	1/1	
PSUBUSW xmmreg1, xmmreg2	66h	0Fh	D9h		Double	FADD/ FMUL	2	1/1	
PSUBUSW xmmreg, mem128	66h	0Fh	D9h		Double	FADD/ FMUL	4	1/1	
PSUBW xmmreg1, xmmreg2	66h	0Fh	F9h		Double	FADD/ FMUL	2	1/1	
PSUBW xmmreg, mem128	66h	0Fh	F9h		Double	FADD/ FMUL	4	1/1	
PUNPCKHBW xmmreg1, xmmreg2	66h	0Fh	68h		Double	FADD/ FMUL	2	1/1	
PUNPCKHBW xmmreg, mem128	66h	0Fh	68h		Double	FADD/ FMUL	4	1/1	
PUNPCKHDQ xmmreg1, xmmreg2	66h	0Fh	6Ah		Double	FADD/ FMUL	2	1/1	
PUNPCKHDQ xmmreg, mem128	66h	0Fh	6Ah		Double	FADD/ FMUL	4	1/1	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
PUNPCKHQDQ xmmreg1, xmmreg2	66h	0Fh	6Dh		Double	FADD/ FMUL	2	1/1	
PUNPCKHQDQ xmmreg, mem128	66h	0Fh	6Dh		Double	FADD/ FMUL	4	1/1	
PUNPCKHWD xmmreg1, xmmreg2	66h	0Fh	69h		Double	FADD/ FMUL	2	1/1	
PUNPCKHWD xmmreg, mem128	66h	0Fh	69h		Double	FADD/ FMUL	4	1/1	
PUNPCKLBW xmmreg1, xmmreg2	66h	0Fh	60h		Double	FADD/ FMUL	2	1/1	
PUNPCKLBW xmmreg, mem128	66h	0Fh	60h		Double	FADD/ FMUL	4	1/1	
PUNPCKLDQ xmmreg1, xmmreg2	66h	0Fh	62h		Double	FADD/ FMUL	2	1/1	
PUNPCKLDQ xmmreg, mem128	66h	0Fh	62h		Double	FADD/ FMUL	4	1/1	
PUNPCKLQDQ xmmreg1, xmmreg2	66h	0Fh	6C		DirectPath	FADD/ FMUL	2	2/1	
PUNPCKLQDQ xmmreg, mem128	66h	0Fh	6C		DirectPath	FADD/ FMUL/ FSTORE	4	2/1	
PUNPCKLWD xmmreg1, xmmreg2	66h	0Fh	61h		Double	FADD/ FMUL	2	1/1	
PUNPCKLWD xmmreg, mem128	66h	0Fh	61h		Double	FADD/ FMUL	4	1/1	
PXOR xmmreg1, xmmreg2	66h	0Fh	EFh		Double	FADD/ FMUL	2	1/1	
PXOR xmmreg, mem128	66h	0Fh	EFh		Double	FADD/ FMUL	4	1/1	
SHUFPD xmmreg1, xmmreg2, imm8	66h	0Fh	C6h		VectorPath	~	4		
SHUFPD xmmreg, mem128, imm8	66h	0Fh	C6h		VectorPath	~	6		
SQRTPD xmmreg1, xmmreg2	66h	0Fh	51h		Double	FMUL	51	1/48	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Table 21. SSE2 Instructions (Continued)

Syntax	Encoding				Decode type	FPU pipe(s)	Latency	Throughput	Note
	Prefix byte	First byte	2nd byte	ModRM byte					
SQRTPD xmmreg, mem128	66h	0Fh	51h		Double	FMUL	53	1/48	
SQRTSD xmmreg1, xmmreg2	F2h	0Fh	51h		DirectPath	FMUL	27	1/24	
SQRTSD xmmreg, mem64	F2h	0Fh	51h		DirectPath	FMUL	29	1/24	
SUBPD xmmreg1, xmmreg2	66h	0Fh	5Ch		Double	FADD	5	1/2	
SUBPD xmmreg, mem128	66h	0Fh	5Ch		Double	FADD	7	1/2	
SUBSD xmmreg1, xmmreg2	F2h	0Fh	5Ch		DirectPath	FADD	4	1/1	
SUBSD xmmreg, mem128	F2h	0Fh	5Ch		DirectPath	FADD	6	1/1	
UCOMISD xmmreg1, xmmreg2	66h	0Fh	2Eh		VectorPath	FADD	4	1/1	
UCOMISD xmmreg, mem64	66h	0Fh	2Eh		VectorPath	FADD	5	1/1	
UNPCKHPD xmmreg1, xmmreg2	66h	0Fh	15h		Double	FADD/ FMUL	2	1/1	
UNPCKHPD xmmreg, mem128	66h	0Fh	15h		Double	FADD/ FMUL/ FSTORE	4	1/1	
UNPCKLPD xmmreg1, xmmreg2	66h	0Fh	14h		DirectPath	FADD/ FMUL	2	2/1	
UNPCKLPD xmmreg, mem128	66h	0Fh	14h		DirectPath	FADD/ FMUL/ FSTORE	4	2/1	
XORPD xmmreg1, xmmreg2	66h	0Fh	57h		Double	FMUL	3	1/2	
XORPD xmmreg, mem128	66h	0Fh	57h		Double	FMUL	5	1/2	

Notes:

1. *The low half of the result is available one cycle earlier than listed.*
2. *This is the execution latency for the instruction. The time to complete the external write depends on the memory speed and the hardware implementation.*

Appendix D AGP Considerations

Fast write transactions are AGP data transfers that originate from processor-issued memory writes. Frequently, the target of fast writes are graphics accelerators and involve:

- Memory-mapped I/O registers (for example, the command FIFO).
- Graphics (2D/3D) engines.
- DVD (motion compensation, sub-picture, etc.) engine registers.
- Frame buffer (render buffers, textures, etc.)

This appendix covers the following topics:

D.1 Fast-Write Optimizations

Fast-write transfers use the PCI addressing semantics but transfer data using the AGP transfer rates (for example, 2x, 4x, or 8x) and AGP flow control between data blocks. The AMD-8151™ HyperTransport™ AGP 3.0 graphics tunnel converts processor memory writes (embedded in HyperTransport traffic) into fast-write transactions on the AGP bus. Fast writes offer an alternative to having the processor place data in memory, and then having the AGP accelerator read the data.

Fast-write transfers are generated to the accelerator with a transfer start address, and then transfer data 32 bits at a time (*start_address* + 0, *start_address* + 4, *start_address* + 8, and so on) until the entire block has been transferred. In this sense, the data is sequential (as it is in DMA). Following are the AGP bus characteristics:

- The AGP bus clock is 66 MHz.
- The AGP data width is 32 bits; at the 8x transfer rate, eight doublewords (32 bytes) can be transferred per AGP clock.

The theoretical data bandwidths for fast writes at 2x, 4x, and 8x are approximately 528 Mbytes/s, 1.056 Gbytes/s, and 2.1 Gbytes/s, respectively. These numbers are theoretical in terms of sustained bursts occurring on the AGP bus. In actuality, data bandwidth depends on the size of the data block transferred from the processor—larger block transfers are better.

Real bandwidth will be lower than the theoretical bandwidth because the beginning of fast-write transactions require sending a PCI-protocol start transaction cycle (for the address phase) at the 1x transfer rate instead of the higher speeds (2x, 4x, or 8x).

Larger block transfers help hide the transaction-start overhead (smaller block transfers have lower bandwidth). For example, at the 8x data-transfer rate, 128 bytes of data can be transferred in four AGP clock cycles, but one initial clock cycle is required for the address phase. Five clock cycles are required to transfer 128 bytes of data; therefore, the overhead of the address phase (clock cycle 1) for 128 bytes of data transferred is 20% (yielding a bandwidth of approximately 1.7 Gbytes/s). See Figure 8.

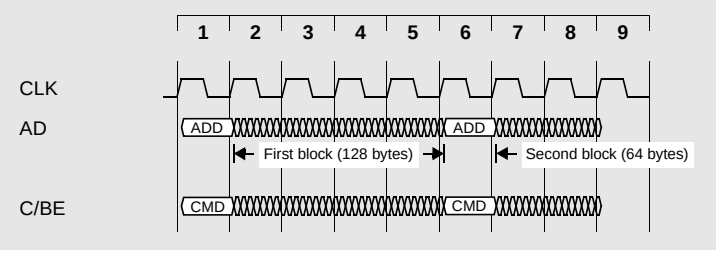


Figure 8. AGP 8x Fast-Write Transaction

The overhead of the address phase for 64 bytes of data is 33% (yielding a bandwidth of approximately 1400 Mbytes/s). For 32 bytes of data (or less), the bandwidth drops to approximately 1000 Mbytes/s. A key software optimization is to buffer as much processor write data as practical.

D.2 Fast-Write Optimizations for Graphics-Engine Programming

Write-combining provides excellent AGP fast-write bandwidth when using the programmed I/O (PIO) model—not the DMA model—for programming 2-D and 3-D graphics engines. To help ensure that data is sent in the most optimal block sizes, “shadow” the engine’s render commands (that is, the registers needed for a render command) in cache-block-aligned data structures in system memory.

Shadowing the structure in system memory (instead of writing the actual write-combining buffer in memory-mapped I/O space) ensures that the write buffer is not emptied prematurely by external events (such as an uncacheable read or hardware interrupt). Shadowing also ensures that writes to different cache lines in the structure do not flush (close) the write-combining buffer since the number of write-combining buffers that can be open at one time is processor-implementation dependent.

On the AMD Athlon™ 64 and AMD Opteron™ processors, write-combining can be used, and software can take advantage of the fact that writes are sent out of the processor's write buffers in ascending order (and appear on HyperTransport that way), from low quadword to high quadword.

Use the Memory Type Range Register (MTRR) mechanism in conjunction with the PAT MSR (model-specific register 277h) to enable write-combining as the memory type for the FIFO address space.

To enable write-combining as the memory type for the FIFO address space, follow these steps:

1. Change the PAT MSR entries that contain a type value of 00h (UC-uncacheable) to a type value of 07h (UC-minus).
2. Program an MTRR with the physical addres and mask range of the command FIFO.

Note: *MTRR registers mark addresses on page granularity boundaries of 4 Kbytes, so the FIFO address should begin on a 4-Kbyte-aligned address boundary).*

For more information, see Chapter 7, “Memory System,” in volume 2 of the *AMD64 Architecture Programmer’s Manual*, order# 24593.

Many graphics engines have a front-end command FIFO that requires the render command to be issued first, followed by a variable number of doublewords, depending on the render command.

Create a cache-aligned command structure in cacheable memory, map the rendering command into the lowest doubleword of the structure (which will be issued first), map the next data required in the command into the next structure element, and so on, until all the data “registers” for this command are included in the structure. An example is given in Figure 9.

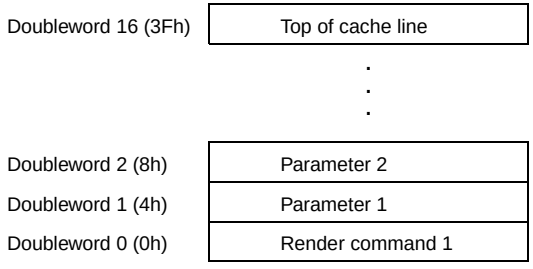


Figure 9. Cacheable-Memory Command Structure

When the command (or commands) are filled in the shadowed structure, use a high-speed copy routine like the one shown in Listing 31 on page 342. Copy the structure to the actual graphic accelerator’s write-combining FIFO address space. Locating the write-combining command FIFO at a cache-aligned address is slightly better, since one HyperTransport link-size write occurs instead of two).

If there are any “empty” doublewords between the last parameter and the top of the cache line, use the SFENCE instruction to flush the write-combining buffer. The data is issued in ascending order. SFENCE is needed to flush the processor’s write-combining buffer on any partially filled buffer. In general, use SFENCE when all parameters needed for rendering have been copied to the memory-mapped I/O (MMIO) FIFO. This ensures that write data is not kept in the processor’s write-combining buffer (which prevents the graphics engine from receiving an incomplete command until the buffer is eventually flushed).

The AGP 3.0 specification specifies that accelerators must be able to buffer at least 128 bytes for the initial data block transferred. Try using 64–128 bytes as the optimal transfer size whenever possible (one to two processor cache lines). Map as many commands as will fit into this 64–128-byte structure.

Listing 31. Sending Write-Combined Data to the Graphics-Engine Command FIFO

```
/* Send commands to a graphic accelerator 2D engine. */
/* The shadowed structure contains 32 DWORDs worth of */
/* rendering commands and data parameters. */
/* Send out 128 (80h) bytes to FIFO in WC MMIO space. */
/* First load 64-bit pointer to a cached command structure. */

mov rdi, OFFSET ShadowRegs_Structure

/* We now have a pointer to the shadowed engine structure. */
/* Grab 16 bytes at a time. */

movdqa xmm0, [rdi]
movdqa xmm1, [rdi + 16]
movdqa xmm2, [rdi + 32]
movdqa xmm3, [rdi + 48]
movdqa xmm4, [rdi + 64]
movdqa xmm5, [rdi + 80]
movdqa xmm6, [rdi + 96]
movdqa xmm7, [rdi + 112]

/* Now get linear pointer to graphic engine mapped in */
/* WC address space. */

mov rax, PTR [Linear2Dengine_Ptr]

/* Now copy register data to processor’s WC buffer. */
/* It is slightly more optimal if the command FIFO */
/* is at a cache-line-aligned address. */
/* Write 16 bytes at a time. */

movdqa [rax], xmm0
movdqa [rax + 16], xmm1
movdqa [rax + 32], xmm2

/* The first WC buffer will be sent after the next write */
/* (assuming FIFO is cache-line aligned) since we’re crossing */
/* a cache-line boundary. */
```

```
movdqa [rax + 48], xmm3

/* Allocate and fill another WC buffer. */

movdqa [rax + 64], xmm4
movdqa [rax + 80], xmm5
movdqa [rax + 96], xmm6

/* The second WC buffer is forced after the next write. */
/* The linear ascending order between cache lines */
/* is maintained since buffer is sent when filled. */

movdqa [rax + 112], xmm7
SFENCE

/* The SFENCE forces the write-combining buffer */
/* out of the processor and to the graphics chip. */
/* Set up the next drawing commands in cached */
/* memory structure ShadowRegs_Structure. */
```

D.3 Fast-Write Optimizations for Video-Memory Copies

When performing block copies of an image to the graphics accelerator's local memory, you can preserve the contents of the L1 and L2 caches and reduce cache-line-replacement traffic to system memory by using a nontemporal block prefetch on the image data using the PREFETCHNTA instruction. This works well with images loaded into system memory through disk DMA because the data can be kept out of the L2 cache and mostly out of the L1 data cache (when using PREFETCHNTA). This is illustrated in Listing 32

Note: *On the AMD Athlon™ 64 and AMD Opteron™ processors, PREFETCHNTA uses one way of the two-way set-associative L1 data cache. One way of the L1 data cache is 32 Kbytes, so limit the block prefetch size to less than or equal to 32 Kbytes.*

Listing 32. Writing Nontemporal Data to Video RAM

```
/* Copy an image larger than 32 Kbytes into local memory, */
/* but limit the block prefetch so as not to exceed 32 Kbytes, */
/* which is the size of the nontemporal cache. */
/* First, block prefetch 16 Kbytes into the L1 data cache, then write */
/* it to the frame buffer. */
/* On AMD Athlon 64 and AMD Opteron processors, the PREFETCHNTA instruction must
execute prior */
/* to subsequent instructions. */
/* Cache lines that are prefetched via PREFETCHNTA and later replaced are */
/* not evicted to the L2 cache or system memory. */
```

```
...

// Use half of the 32-Kbyte nontemporal cache for a block load.
#define HALFL1PREFETCHNTACACHESIZE 16384

mov rdi, QWORD PTR [image_source]
mov rcx, HALFL1PREFETCHNTACACHESIZE / 64

Block_PrefetchIntoL1:
prefetchnta QWORD PTR [rdi]   ; Grab 64 bytes.
add         rdi, 64           ; Bump up to next cache line.
dec rcx
jnz         Block_PrefetchIntoL1

LoadPtr_ToFrameBuffer:
mov rdi, QWORD PTR [frameBuffDestPtr]
mov rcx, HALFL1PREFETCHNTACACHESIZE / 128

/* Get linear pointer to local memory mapped in WC address space. */

mov rax, DQWORD PTR [FBimage_Ptr]

/* Send out 128 bytes (yielding ~1.7 Gbytes/s of fast-write bandwidth) */
/* per block. RDI now has pointer back to image source. */
/* 16 Kbytes of image is in L1 nontemporal cache (way 0 of cache). */

Block_WriteToFrameBuffer:
movdqa xmm0, [rdi]
movdqa xmm1, [rdi+16]
movdqa xmm2, [rdi+32]
movdqa xmm3, [rdi+48]
movdqa xmm4, [rdi+64]
movdqa xmm5, [rdi+80]
movdqa xmm6, [rdi+96]
movdqa xmm7, [rdi+112]

/* Copy register data to WC buffer. */

movdqa [rax], xmm0
movdqa [rax+16], xmm1
movdqa [rax+32], xmm2

/* The first WC buffer is sent after next write since we're crossing */
/*  a cache-line boundary. */

movdqa [rax+48], xmm3

/* Allocate and fill another WC buffer. */

movdqa [rax+64], xmm4
movdqa [rax+80], xmm5
```

```
movdqa [rax+96], xmm6
movdqa [rax+112], xmm7
add rax, 128   ; Bump up by 2 cache lines
add rdi, 128   ;  for source and destination.
dec rcx
jnz Block_WriteToFrameBuffer

ChunkOfImageCopied:

/* Set up for next block in image (if necessary) */
/* until image is transferred. */
```

D.4 Memory Optimizations

AGP memory is system memory that is partitioned from the same memory that the operating system and applications use. The AGP card plugged into the AGP bus is always considered the master when performing AGP memory accesses since it reads and writes the system memory. The AGP card uses AGP memory for a variety of “surfaces,” including:

- Texture maps
- 3-D object geometry and vertex data streams
- Command buffers for 2-D and 3-D graphics engines
- Video-capture buffers
- Frame buffer (cost-reduced implementations)

The system memory used for AGP mastering is attached to the processor that has one of its HyperTransport links connected to an AGP tunnel device, such as the AMD-8151 HyperTransport AGP 3.0 graphics tunnel. AGP card requests (reads/writes) come into the processor through the HyperTransport link input and are arbitrated with processor requests for system memory in the system request queue (SRQ). From here, the AGP request address is passed into the processor’s address map and GART (graphics aperture remapping table), where the AGP physical address is translated into a physical DRAM page address, which can then be presented to the processor’s memory controller. Therefore, host processor to system memory throughput directly affects AGP memory bandwidth and throughput, as the two compete for SRQ entries and memory bandwidth. Figure 10 shows the command flow from the HyperTransport links to the SRQ.

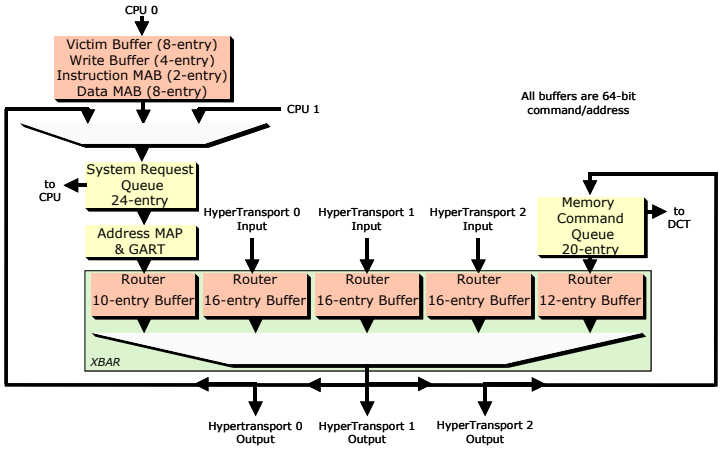


Figure 10. Northbridge Command Flow

D.5 Memory Optimizations for Graphics-Engine Programming Using the DMA Model

Historically (that is, with AGP 1.0 and AGP 2.0), AGP memory used for command DMA buffers was accessed by the processor through the AGP aperture space (this feature is referred to as *host translation*). This address space was mapped as write-combining due to the fact that the processor's caches were not snooped by an AGP master (that is, coherency was not enforced for AGP memory). Write-combining offered the best bandwidth in this situation because write-combining buffers could be sent to system memory as full write-combining buffers. However, system memory still needed to be written, which used memory bandwidth.

On current systems however, coherency between an AGP master (making accesses through the AGP aperture) and the processor caches is maintained due to the HyperTransport protocol and the MOESI (modified, owner, exclusive, shared, invalid) caching policy. Coherency support between an AGP master and the processor caches is enabled through a bit in the GART entry (Gart_entry.coh). The AGP miniport driver sets this bit as it maps entries in the GART. The video graphics miniport driver can verify this feature in the AGP 3.0-compliant register (AGPSTAT.ita_entry.coh), which is found in the AGP bridge device.

Note: *Coherency support is implemented by hardware in AMD Athlon 64 and AMD Opteron processors, and is not specific to the AGP tunnel device, even though the support is indicated in the tunnel's AGP 3.0-compliant register (AGPSTAT.ita_entry.coh).*

Therefore, a key optimization for the DMA model on AMD Athlon 64 and AMD Opteron processors is that the AGP master may read the data from the processor caches faster than reading data from the DDR memory, since the processor caches operate at higher clock frequencies. As processor clock

frequencies increase, so will the ratio of operating frequencies between processor caches and DDR memory. The processor-to-write-back cache bandwidth is also higher than processor-to-AGP-aperture bandwidth (write-combining memory type), since the DDR writes are avoided (as well as GART translation latencies).

It may be possible to prevent pollution of the L1-data and L2 caches from DMA data by using the nontemporal PREFETCHNTA instruction on the DMA buffer and limiting prefetching of the DMA buffer to less than 32 Kbytes (PREFETCHNTA uses only one way of the L1 data cache).

Use PREFETCHNTA on the linear address to the DMA buffer, and not the AGP aperture address, before reading or writing the DMA buffer.

Another key optimization for the DMA model on AMD Athlon 64 and AMD Opteron systems is that coherency is maintained between processor caches and an AGP master making accesses outside of the AGP aperture.

This is a key AGP enhancement that is required of AGP 3.0 target (host platform) systems.

In effect, this means that an AGP master can create a DMA buffer in normal write-back memory and then pass the physical DRAM page address to the AGP master; in other words, the AGP virtual address and GART translation is not used.

Use PREFETCHNTA on the linear address to the DMA buffer, before reading or writing the DMA buffer.

If the AGP card hardware is capable of buffering the physical DRAM page addresses sent to the AGP card in a FIFO, then in effect the AGP card's device driver is getting AGP scatter-gather capabilities, with cache coherency provided by the processor.

D.6 Optimizations for Texture-Map Copies to AGP Memory

To avoid cache pollution, use the same technique described in "Fast-Write Optimizations for Video-Memory Copies" on page 343 to copy texture data into AGP memory, since this data tends to be nontemporal.

D.7 Optimizations for Vertex-Geometry Copies to AGP Memory

To avoid cache pollution, use the same technique described in "Fast-Write Optimizations for Video-Memory Copies" on page 343 to copy vertex data into AGP memory, since this data tends to be nontemporal.

Appendix E SSE and SSE2 Optimizations

Certain versions of the AMD Opteron™ processor can benefit from additional optimizations when using SSE and SSE2 instructions. This appendix describes those optimizations. These coding suggestions apply only to revision B processors. Later revisions of the processor do not need these optimizations. For revision information, see the *Revision Guide for AMD Opteron™ Processors*, order# 25759.

Types of XMM-Register Data

The XMM registers (used by the SSE and SSE2 instructions) can hold the following three types of data:

- Floating-point single-precision (FPS)
- Floating-point double-precision (FPD)
- Integer (INT)

Types of SSE and SSE2 Instructions

Most SSE and SSE2 instructions can be divided into five types according to the type of data they produce and therefore expect to consume:

- Floating-point single-precision (FPS)
- Floating-point double-precision (FPD)
- Integer (INT)
- Load (produces data of type FPS, FPD, or INT)
- Store (can consume a register with data of any type)

This appendix covers the following topics:

E.1 SSE and SSE2 Instruction and Data Types

Optimization

❖Make sure that data to be consumed by an SSE or SSE2 instruction is of the same type as the instruction. (For definitions of these types, see "Types of XMM-Register Data" on page 349 and "Types of SSE and SSE2 Instructions" on page 349.)

To use an SSE or SSE2 instruction of this type	Make sure the data it will consume is of this type
FPS	FPS
FPD	FPD
INT	INT
Store	Any type (FPS, FPD, or INT)

Since load instructions do not consume a register, they are not listed in the table. Similarly, instructions that do an implicit load, such as `addsd xmmreg, mem64`, do not require this type matching.

Note: *All INT loads (MOVDQA for 128 bits and MOVQ for 64 bits) leave the register in a format that is acceptable to all SSE and SSE2 instructions.*

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

If the data to be consumed by an instruction is not of the expected type, the processor may require additional time to execute the instruction.

Example 1

Avoid code like this:

```
movsd  xmm4, xmm0       ; Move a scalar double (FPD) into XMM4.
pextrw ecx, xmm4, 03h   ; Extract a 16-bit (INT) value from XMM4.
```

In this example, the first instruction sets the XMM4 register's data type to FPD. The PEXTRW instruction, however, expects XMM4 to be in INT format.

Example 2

Avoid code like this:

```
subpd  xmm4, [004147e0h]   ; Subtract double-precision values.
pshufd xmm4, xmm4, eeh     ; Shuffle 32-bit integer values.
```

In this example, the SUBPD instruction sets the XMM4 register's data type to FPD. The PSHUFD instruction, however, expects XMM4 to be in INT format.

Half-Register Operations

Attention should be paid to mixed data types even within the same register. For example:

```
addps    xmm1, xmm2
cvtss2sd xmm1, xmm2
```

Here, the last instruction leaves the upper half of XMM1 in FPS format and the lower half in FPD format.

Another example is:

```
addps    xmm1, xmm2
movlpd   xmm1, mem64
```

The MOVLPD instruction sets the low half of XMM1 to FPD format but leaves the high half unchanged (in FPS format).

Mixing data types in a single register is harmless if only scalar operations are used. However, this practice can cause performance problems if the register is used as a source for a vector operation (such as XMM2 in `movaps xmm1, xmm2` or `movapd xmm1, xmm2`).

Some instructions (for example, logical operators like AND, OR, and ANDN) only exist in vector form. When such instructions are used, make sure that both halves of both source registers are the required type.

Zeroing Out an XMM Register

When an XMM register needs to be zeroed out, it should be done with an instruction whose format matches the format required by the consumers of the zeroed register. Table 22 on page 353 shows the different possible consumers of an XMM register and the corresponding instruction that should be used to zero out the register.

Table 22. Clearing XMM Registers

Producer of Zero	Example Consumers of Zero
xorpd xmm1, xmm1	cmppd xmm1, xmm2
	cmpsd xmm1, xmm2
	comisd xmm1, xmm2
	maxpd xmm1, xmm2
	maxsd xmm1, xmm2
	ucomisd xmm1, xmm2
	subsd xmm1, xmm2
xorps xmm1, xmm1	cmpps xmm1, xmm2
	cmpss xmm1, xmm2
	comiss xmm1, xmm2
	maxps xmm1, xmm2
	maxss xmm1, xmm2
	ucomiss xmm1, xmm2
	subss xmm1, xmm2
pxor xmm1, xmm1	pcmpxxx xmm1, xmm2
	pmaxxx xmm1, xmm2
	psubxxx xmm1, xmm2

E.2 Bit Manipulations on Floating-Point Numbers

Optimization

Using either the XMM registers or the general-purpose registers to access the binary representation of a floating-point number and examine its sign, exponent, or mantissa.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale—Using the General-Purpose Registers

Transferring a floating-point number from an XMM register to a general-purpose register can cause a loss in performance. A preferred way to prevent this loss in performance is to avoid a direct register-to-register transfer. In particular, avoid the use of MOVD and PEXTRW for moving FPS or FPD data from the XMM registers to the general-purpose registers. Instead, load the GPR directly from memory. If the floating-point number is only in an XMM register, then the preferred alternative is to write it out to memory from the XMM registers and then load it from memory into the general-purpose registers.

Rationale—Using the XMM Registers

Bit manipulation of floating-point values in XMM registers—particularly logical shifts—may be less than optimal. However, if bit manipulation must be done, use integer SSE or SSE2 ALU instructions and loads from memory to minimize any negative performance impact.

E.3 Reuse of Dead Registers

Optimization

When it is necessary to save the contents of a register that is in FPS format to another unused (or *dead*) register, where the previous contents of the dead register are unknown, then use `movaps xmm1, xmm2` instead of `movss xmm1, xmm2`.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The `movss xmm1, xmm2` instruction takes additional time to execute if the previous contents of XMM1 are not in FPS format.

The performance of `movaps xmm1, xmm2` can suffer if part of the XMM2 register is not in FPS format (see "Half-Register Operations" on page 352). For this reason, it's best to avoid instructions like `movlps xmm1, mem64` that set only the lower half of an XMM register to FPS format.

E.4 Moving Data Between XMM Registers and GPRs

Optimization

Store a register that needs to be spilled in memory, rather than moved to a different register file.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

While register moves within a given register file are very efficient (XMM to XMM, GPR to GPR), moves between register files (XMM to GPR, GPR to XMM) are not. .

E.5 Saving and Restoring Registers of Unknown Format

Optimization

Use INT loads (MOVDQA for 128 bits and MOVQ for 64 bits) when restoring registers of unknown format from the stack.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

All stores of 64-bits or more from an XMM register to memory may be performed without concern for the type of the data in the XMM register. This allows called procedures to save registers on the stack without knowing what their format was. Conversely, all INT loads (MOVDQA for 128 bits and MOVQ for 64 bits) leave the register in a format that is acceptable to all SSE and SSE2 instructions and is recommended when restoring registers of unknown format from the stack.

E.6 SSE and SSE2 Copy Loops

When copying data of an unknown format using the XMM registers, it is best to use INT loads and stores.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When using SSE and SSE2 instructions to perform loads and stores, it is best to interleave them in the following pattern—Load, Store, Load, Store, Load, Store, etc.

If in 32-bit mode and using MMX instructions to perform loads and stores, they should be arranged in the following pattern—Load, Load, Store, Store, Load, Load, Store, Store, etc.

Example

The following example illustrates a sequence of 128-bit loads and stores:

```
movdqa      xmm0, [rdx+r8*8]       ; Load
movntdq     [rcx+r8*8], xmm0       ; Store
movdqa      xmm1, [rdx+r8*8+16]    ; Load
movntdq     [rcx+r8*8+16], xmm1    ; Store
```

E.7 Explicit Load Instructions

Optimization

Use `movlpd xmm1, mem64` when loading a scalar FPD value from memory.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The `movlpd xmm1, mem64` instruction is more efficient than `movsd xmm1, mem64`. Use MOVSD only if you need to ensure that the upper half of XMM1 is also set to FPD format, perhaps because a vector operation is planned on the register.

When loading a scalar FPS value from memory, use MOVSS.

E.8 Data Conversion

Optimization

Use care when selecting instructions to convert values from one type to another.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

For example, the CVTDQ2PS instruction converts four packed 32-bit signed integer values in an XMM register or a 128-bit memory location to four packed single-precision floating-point values and writes the converted values to another XMM register. In some cases, an additional instruction is recommended to ensure that both halves of register operands are of the same type (as recommended in “Half-Register Operations” on page 352).

Table 23 shows the recommendations for register-to-register conversion of scalar values. Table 24 on page 361 shows the recommendations for register-to-register conversion of vector operands. When converting values directly from memory, use the preferred instructions provided in Table 25 on page 361.

Table 23. Converting Scalar Values

Source format	Destination format	Preferred instructions	Notes
FPS	INT XMM	`cvtps2dq xmm1, xmm2`	
FPS	INT GPR	`cvtss2si reg32/64, xmm1`	
FPS	FPD	`cvtss2sd xmm1, xmm2`	
FPD	INT XMM	`unpcklpd xmm2, xmm2` `cvtpd2dq xmm1, xmm2`	UNPCKLPD ensures that the high half of XMM2 is also in FPD format.
FPD	INT GPR	`cvtsd2si reg32/64, xmm1`	
FPD	FPS	`xorps xmm1, xmm1` `cvtsd2ss xmm1, xmm2`	XORPS ensures that the high half of XMM1 is in FPS format in case a MOVAPS instruction is used later.
INT XMM	FPS	`cvtdq2ps xmm1, xmm2`	
INT XMM	FPD	`cvtdq2pd xmm1, xmm2`	

Table 23. Converting Scalar Values (Continued)

Source format	Destination format	Preferred instructions	Notes
INT GPR	FPS	`xorps xmm1, xmm1` `cvtsi2ss xmm1, reg32/64`	XORPS is used to ensure that the high half of XMM1 is in FPS format. This is also better in case a MOVAPS instruction is used later.
INT GPR	FPD	`cvtsi2sd xmm1, reg32/64`	

Table 24. Converting Vector Values

Source format	Destination format	Preferred instructions	Notes
FPS	INT XMM	`cvtps2dq xmm1, xmm2`	
FPS	FPD	`cvtps2pd xmm1, xmm2`	
FPD	INT XMM	`cvtpd2dq xmm1, xmm2`	
FPD	FPS	`cvtpd2ps xmm1, xmm2`	
INT XMM	FPS	`cvtdq2ps xmm1, xmm2`	
INT XMM	FPD	`cvtdq2pd xmm1, xmm2`	

Table 25. Converting Directly from Memory

Source format	Destination format	Preferred instructions	Notes
FPD	FPS	`xorps xmm1, xmm1` `cvtsd2ss xmm1, mem64`	XORPS ensures that the high half of XMM1 is in FPS format in case a MOVAPS instruction is used later.
INT GPR	FPS	`xorps xmm1, xmm1` `cvtsi2ss xmm1, mem32/64`	XORPS is used to ensure that the high half of XMM1 is in FPS format. This is also better in case a MOVAPS instruction is used later.

E.9 Comparisons and Logical Operations on Floating-Point Numbers

Optimization

Use the output of scalar and vector versions of the SSE comparison instructions only with other logical operators of the same floating-point type.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The comparison instructions CMPSS and CMPPS produce FPS results. Similarly, CMPSD and CMPPD produce FPD results. This is true even though the output looks like an INT value instead of a floating-point number. Therefore, use the output of these instructions only with other logical operators of the same floating-point type.

There are both scalar and vector variants of SSE comparison instructions such as CMPSS and CMPPS, but there are only vector versions of the AND, ANDN, OR, and XOR logical operators. For this reason—and to achieve the best performance—make sure both halves of the registers are of the correct type (FPS for packed single-precision instructions and FPD for packed double-precision instructions) when using the vector instructions .

E.10 Swizzling from Memory

Building a vector XMM register from disjoint memory locations is sometimes referred to as *swizzling*. For best performance, the following recommendations should be observed:

Swizzling FPS Values

When swizzling FPS values using UNPCKLPS, UNPCKHPS, or SHUFPS, the full destination and full source should be of type FPS. Each of the following instructions is useful for setting the destination register to full FPS format:

- `movss  xmm1, mem32`
- `movaps xmm1, mem128`
- `movaps xmm1, xmm2     ; Note: XMM2 should also be fully FPS.`
- `xorps  xmm1, xmm1`

Swizzling FPD Values

When swizzling FPD values using UNPCKLPD, UNPCKHPD, or SHUFPD, the full destination and full source should be of type FPD. Each of the following instructions can be used to set the destination register to full FPD format:

- `movsd  xmm1, mem64`
- `movapd xmm1, mem128`
- `movapd xmm1, xmm2     ; Note: XMM2 should also be fully FPD.`
- `xorpd  xmm1, xmm1`

Swizzling INT Values

Swizzle INT values using the PUNPCK*xxx* instructions. For PUNPCKL*xx*, the low parts of both the source and destination should be of type INT. For PUNPCKH*xx*, the high part of both the source and destination should be of type INT. Each of the following instructions can be used to prepare a register:

- `movq   xmm1, mem64`
- `movdqa xmm1, mem128`
- `movdqa xmm1, xmm2     ; Note: XMM2 should also be fully INT.`
- `pxor   xmm1, xmm1`

Swizzling Word Values

Use PINSRW to swizzle word values. The full destination should be of type INT for best performance. Before doing the first PINSRW on a register of unknown type, use `pxor xmm1, xmm1` to force it to INT format.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Index

Numerics

A

B

C

D

E

F

I

L